# Fail-safe Mobility Management and Collision Prevention Platform for Cooperative Mobile Robots with Asynchronous Communications

by

Rami YARED

**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*

September 2006

# Abstract

Distributed computing extends its scope to address problems relevant to mobile computing where hosts are physically mobile. Since a robot can be seen as a mobile computer, it is natural to consider a group of autonomous mobile robots as a kind of mobile distributed system. However, there are two fundamental differences with conventional distributed systems. The first is that robots usually require knowledge about their physical positions, and the second is that robots must control their own motion.

Many interesting applications of mobile robotics envision groups or swarms of robots cooperating toward a common goal. Consider a distributed system composed of cooperative autonomous mobile robots cultivating a garden. This application requires that robots move in all directions sharing the same geographical space. We consider a category of robotic applications where mobile robots have limited energy resources and wide geographical distribution. There is no centralized control nor global synchronization.

It is very important to focus on the problem of preventing collisions between mobile robots. Collision prevention leads to a dependable system and prevents the occurrence of serious damages to the robots which causes failures in the system.

In order to achieve a fail-safe motion, robots need to coordinate their movement. Cooperation is however difficult to obtain under the weak communication guarantees offered by wireless networks, because retransmission of messages is needed to ensure messages delivery in wireless environments. The communication delays to deliver messages are difficult to anticipate. Therefore, a time-free collision prevention protocol is very important in wireless environments.

The main contribution of this dissertation is providing a motion coordination platform that makes a system of mobile robots fail-safe independently of timeliness properties of the system. Mobile robots rely on this platform for their motion planning. The mobility coordination platform consists of time-free collision prevention protocols for an asynchronous system of cooperative mobile robots. The platform guarantees that no collision between robots can occur. In this dissertation, we analyze the performance of the protocols. A performance analysis provides insights for a proper dimensioning of system's parameters in order to maximize the average effective speed of robots. We consider also the collision prevention in presence of robots failures by crash, and provide fault-tolerant collision prevention protocols that tolerate the crash of a certain number of robots. We consider two system models, closed group and dynamic group models.

The first contribution is to provide collision prevention protocols for asynchronous cooperative mobile robots in a dynamic group model. In this model, the composition of the system of which robots have only a partial knowledge, can change dynamically. Robots have limited communication range, hence they naturally form an ad hoc network on which they rely for their communication. The collision prevention protocol relies on a Neighborhood Discovery primitive which is readily available through most of wireless communication devices. The collision prevention protocol is based on a locality-preserving distributed path reservation system that takes advantage of the inherent locality of the problem, in order to reduce communication.

The second contribution of this dissertation is to provide collision prevention protocols for asynchronous cooperative mobile robots in a closed group model, in which a robot knows the composition of the group and can always communicate with all robots of the group.

The third contribution is providing group membership and view synchrony protocols among robot teams, in a distributed system model composed of a group of teams of worker robots that rely on physical robot messengers for the communication between the teams. The protocols tolerate the crash of a certain number of messengers robots and teams. Unlike traditional distributed systems, there is a finite amount of messengers in the system, and thus a team can send messages to other teams only when some messenger robot is available locally.

# Acknowledgments

I wish to express my sincere gratitude to my supervisor Prof. Xavier Défago of Japan Advanced Institute of Science and Techology for his constant encouragement and guidance during the three years of my Ph.D. in his Laboratory. His confidence in me and his help in clarifying the results of my work have been important factors to the success of this research.

I wish to express my thanks to Prof. Takuya Katayama of Japan Advanced Institute of Science and Techology for his constant encouragement and helpful discussions.

I am also grateful to Matthias Wiesmann for his helpful collaboration and support throughout my Ph.D. research, as well as his friendship.

I would like to thank Julien Cartigny, Nikolaos Galatos, and Péter Urbán for always being available to discuss different research issues, as well as their friendship.

I am grateful to the members of the jury, Prof. Nak Young Chong, Prof. Yoshiaki Kakuda, Prof. Takuya Katayama, Prof. Yasuo Tan and Prof. Tatsuhiro Tsuchiya for their interesting ideas and reviewing my PhD thesis.

I would also to thank the Japan Ministry of Education, Culture, Sports, Science and Technology, for the financial support of my Ph.D. research for the program "Fostering Talent in Emergent Research Fields" in Special Coordination Funds for Promoting Science and Technology.

My special thanks go to Samia Souissi for her constant encouragement, support, and nice friendship.

I am also grateful to Yasser Kotb, Ahlem Ben Hassine, Shafik and Zoubaida Ansari, Oussama Alhalabi, Adel Kafri, Bachar Mohamad, Amjad Alhalwani and Houssein Jad for their encouragement and friendship.

I would also like to express my gratitude to all the people in Dependable Distributed Systems Group and Katayama Laboratory for their support and friendship, in particular Misato Morita and Miyuki Sakurai for their help with all administrative issues.

I am grateful to Tanizaki Hiroaki for his help to translate the abstract of my dissertation in Japanese. I would also like to thank Maria Gradinariu, Daiki Higashihara, Tomokazu Matsushita, Naohiro Hayashibara, Guenho Lee, Kim Myungsik, Meinin Kou and Miyashita Kanae for their support and help.

I would like to thank: Walid Basha, Mary Ewald, Sally Fischer, Steve Gates, as well as the priests of Hirosaka Catholic Church in Kanazawa for their friendship and support.

I wish to express my gratitude to my teachers in particular the professors of Japan Advanced Institute of Science and Technology (JAIST) in Japan, as well as the professors of Laboratoire d'Analyse et d'Architecture des Systemes (LAAS-CNRS) and Ecole Nationale Supérieure des Télécommunications (ENST-PARIS) in France, and the professors of Institut Superieur des Sciences Appliquées et de Technologie (ISSAT) in Syria.

Last but not least, I devote my sincere thanks to my parents, uncles, brother and sister for their loving support throughout my life.

# Contents

# List of Figures

# Chapter 1

# Introduction

**Context** Distributed computing extends its scope to address problems relevant to mobile computing where hosts are physically mobile. Since a robot can be seen as a mobile computer, it is natural to consider a group of autonomous mobile robots as a kind of mobile distributed system. However, there are two fundamental differences with conventional distributed systems. The first is that robots usually require knowledge about their physical positions, and the second is that robots must control their own motion. [9]

There has been increased research interest in systems composed of multiple autonomous mobile robots exhibiting cooperative behavior. Such systems are of interest for several reasons. Tasks may be inherently too complex (or impossible) for a single robot to accomplish, or performance benefits can be gained from using multiple robots. [5].

Many interesting applications of mobile robotics envision groups or swarms of robots cooperating toward a common goal. Consider a distributed system composed of cooperative autonomous mobile robots exploring an unknown environment [4]. Exploring an unknown environment by cooperative mobile robots requires that mobile robots move in all directions in the same geographical space.

A robot computes a collision-free path, between the current robot location and the goal. This path avoids collisions with fixed obstacles due to motion planning. A robot that operates in an unknown environment, sense directly within the motion. In many classes of applications of autonomous cooperative mobile robots (e.g., exploring unknown environments), where speeds of robots are unknown to other robots, the motion planning approaches cannot guarantee a *safe* motion as mobile robots may collide with each other, because of the unknown speeds of robots and the uncertainty of the sensory information.

We consider a category of robotic applications (e.g., exploration of unknown environments) where mobile robots have limited energy resources and wide geographical distribution. The robots use sensors to explore the environment, they are not provided with a vision capability. In the considered system, there is no centralized control nor global synchronization.

**Motivation** The robots are moving in different directions to explore the environment. Robots share the physical space, thus collisions between mobile robots can possibly occur. It is very important to focus on the problem of *preventing* collisions between the mobile robots. Collision prevention leads to a dependable system and prevents the occurrence of serious damages to the robots which causes failures in the system.

Existing techniques that avoid collisions between mobile robots or vehicles are based on real-time approaches, or assuming the existence of a known constant upper bound on the communication delays, processing speed and on robots speed movement.

A robot knows neither the positions of other robots nor their destinations precisely at a given instant. Additionally, the speed of a robot is unknown by robots and there is no known upper bound on robot's speed since in the considered applications a robot is autonomous and possibly it does not know the composition of the group of robots. So a robot cannot estimate the position of another robot in the system. Therefore, robots need to cooperate in order to coordinate their movement and hence to achieve a fail-safe motion. Cooperation is however difficult to obtain under the weak communication guarantees offered by wireless networks, because retransmission of messages is needed to ensure messages delivery in wireless environments. The communication delays to deliver messages of arbitrary size are difficult to anticipate. Hence, a time-based protocol possibly fails because of violating some timing property in such environments, so, a time-free collision prevention protocol is very important.

**Contribution** The main contribution of this dissertation is providing a motion coordination platform that makes a system of mobile robots fail-safe. Cooperative mobile robots rely on this platform the for their motion planning. The mobility coordination platform consists of time-free collision prevention protocols for an asynchronous system of cooperative mobile robots. The platform guarantees that no collision between robots can occur. In this dissertation, we analyze the performance of the protocols. A performance analysis provides insights for a proper dimensioning of system's parameters in order to maximize the average effective speed of robots.

We consider the collision prevention problem in face of robots failures by crash, and provide fault-tolerant collision prevention protocols that tolerate the crash of some robots and allows the system of robots to progress.

This dissertation provides time-free collision prevention protocols, for asynchronous cooperative mobile robots in two system models, closed group and dynamic group models.

The first contribution is to provide collision prevention protocols for asynchronous cooperative mobile robots in a dynamic group model. In this model, the composition of the system of which robots have only a partial knowledge, can change dynamically. Robots have limited communication range, hence they naturally form an ad hoc network on which they rely for their

communication. The collision prevention protocol relies on a Neighborhood Discovery primitive which is readily available through most of wireless communication devices. The collision prevention protocol is based on a locality-preserving distributed path reservation system that takes advantage of the inherent locality of the problem, in order to reduce communication.

The second contribution of this dissertation is to provide collision prevention protocols for asynchronous cooperative mobile robots in the closed group model, in which a robot knows the entire composition of the group and can always communicate with all robots of the group.

A third contribution is providing group membership and view synchrony protocols among robot teams, in a distributed system model composed of a group of teams of worker robots that rely on physical robot messengers for the communication between the teams. The protocols tolerate the crash of a certain number of messengers robots. Unlike traditional distributed systems, there is a finite amount of messengers in the system, and thus a team can send messages to other teams only when some messenger robot is available locally.

## 1.1 Related work

Martins et al. ([20, 21]) demonstrated a scenario of three cooperating cars, elaborated in the CORTEX project, and relies on the existence of Timely Computing Base (TCB) wormholes. The TCB concept was introduced in ([31, 32]). In [21], authors presented how to use an application's fail-safety and time-elasticity characteristics to overcome the uncertainty of the environment in TCB based systems.

Our approach in Chapter 6 for a dynamic group model and that in [21] use limited local communication and their designs rely on the concept of a *wormhole*. The wormhole of the system in [21] is encapsulated in the TCB components which are interconnected by a *control* network. The control network is isolated from the *payload* network (of the application) using a dual network architecture. The wormhole of our platform is encapsulated in the primitive Neighborhood Discovery which is available through most wireless communication devices. The *fundamental* difference between our fail-safe platform and [21], is that the approach in [21] is time-elastic, while our approach is time-free.

The approach in [21] requires that the composition of the group is known by all the participants, so if we compare the approach in [21] with our approach for the closed group model then, a fundamental difference is that the approach in [21] is based on a wormhole but our approach for the closed group model is purely asynchronous.

Both, our approach for the dynamic group model and the approach in [21] rely on a wormhole, but our dynamic group model does not require that a robot knows the composition of the group, while the approach in [21] requires that the composition of the group is known to each

robot and a robot can communicate with all robots of the group.

Nett et al. ([26, 25]) presented a layered system architecture for cooperative mobile systems in real-time applications. They considered a traffic control application as a testbed of their system architecture. In this testbed a group of mobile robots are driving along two overlapping closed loops sharing a specified predetermined space. The architecture in ([26, 25]) aimed at real-time cooperative mobile systems. Our approach fundamentally differs in several aspects. The system in ([26, 25]) is synchronous assuming the existence of a known constant upper bound on the communication delays, while our approach is asynchronous. The communication infrastructure in ([26, 25]) is based on wireless LAN and the designed communication protocols use the access point (base station) as a central router since each station must be within the reach of the access point, which implies a full connectivity. The mobile robots in our system form naturally a mobile ad hoc network on which they rely for their communication. MANETs have no centralized control nor global synchronization, also the real-time constraints to deliver messages of arbitrary size are not guaranteed.

The problem of robots collision avoidance has been handled using different strategies which are sensor-based motion planning methods. The detailed information about motion planning strategies is inspired from [22].

Minguez et al. [22] compute collision-free motion for a robot operating in dynamic and unknown scenarios, also they survey the existing collision avoidance navigation approaches. Motion planning algorithms consider a model of the environment (either previously known or dynamically built), to compute a collision free path between the current robot location and the goal. In dynamic or unknown environments the trajectories generated by motion planning algorithms become inaccurate thus they can not be applied to such environments. Solving this problem involves sensing directly within the motion planning by applying a *perception-action* process that is repeated periodically at a high rate. These approaches use a local fraction of the information available (sensory information), so they can fall into trap situations. Some of these approaches apply mathematical equations to the sensory information and the solutions are transformed into motion commands. (e.g., [23]). Another group of methods compute a set of suitable motion commands to select one command based on navigation strategy (e.g., [29]), while other methods (e.g., [22]), compute a high level information description (entities near obstacles, areas of free space), from the sensory information then apply different techniques simplifying the difficulty of the navigation to obtain a motion command in complex scenarios.

**Roadmap for this dissertation** This dissertation is organized as follows.

Chapter 2 introduces formal definitions of some agreement problems in distributed systems that this thesis is concerned with, such as the Total order broadcast, failure detectors and their

classes, consensus.

Chapter 3 explains the two system models. the dynamic group and the closed group models. Also, it presents our definitions and terminology which are related to the collision prevention protocols.

Chapter 4 defines the collision prevention problem and presents the properties of the collision prevention protocols.

Chapter 5 presents the collision prevention protocols for the closed group model. It presents the failure-free protocol and proves that it satisfies the properties of the collision prevention problem. A performance analysis is provided and showed that a proper dimensioning of system's parameters in order to maximize the average effective speed of robots.

Also, Chapter 5 introduces Fault-tolerant collision prevention protocols that consider the crash of some robots.

Chapter 6 presents the collision prevention protocol for a dynamic group of asynchronous cooperative mobile robots. Chapter 6 also proves that the collision prevention protocol for a dynamic group satisfies the properties mentioned in Chapter 4 shows that a proper dimensioning of system's parameters in order to maximize the average effective speed of robots.

Chapter 7 provides group membership and view synchrony protocols among robot teams, in a distributed system model composed of a group of teams of worker robots that rely on physical robot messengers for the communication between the teams.

Chapter 8 recalls the main contributions.

# Chapter 2

# Background

## 2.1 Motion planning

Motion planning is defined by finding a route to a robot from an initial position to a final position, in presence of obstacles. The motion planning is illustrated in Figure.2.1

Motion planning algorithms consider a model of the environment (either previously known or dynamically built), to compute a collision free path between the current robot location and the goal. In dynamic or unknown environments the trajectories generated by motion planning algorithms become inaccurate thus they can not be applied to such environments. Solving this problem involves sensing directly within the motion planning by applying a *perception-action* process that is repeated periodically at a high rate. These approaches use a local fraction of the information available (sensory information), so they can fall into trap situations. Some of these approaches apply mathematical equations to the sensory information and the solutions are transformed into motion commands. (e.g., [23]).

## 2.2 Total Order Broadcast

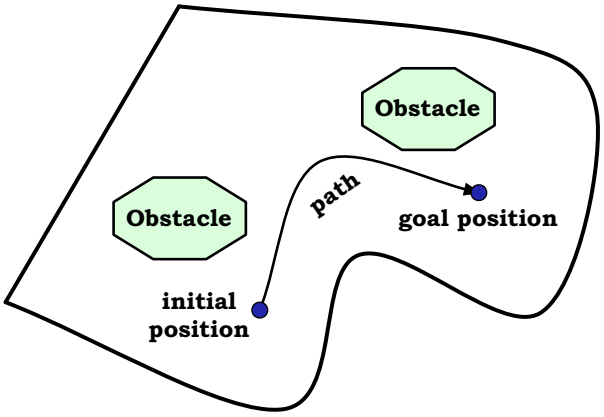



Figure 2.1: Motion planning.

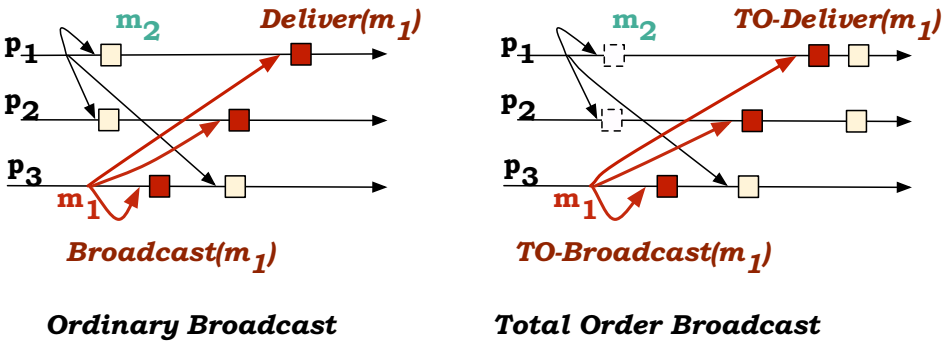



Figure 2.2: Total Order Broadcast primitive.

TOTAL ORDER BROADCAST also called ATOMIC BROADCAST, is a fundamental problem in distributed systems, especially with respect to fault-tolerance. The TOTAL ORDER BROADCAST primitive ensures that messages sent to a set of processes are, in turn, delivered by all those processes in the same total order. Informally, the problem is defined as a broadcast primitive whereby all processes deliver the same sequence of messages. Figure. 2.2 illustrates the total order broadcast primitive.

There exists a vast amount of literature about Total Order Broadcast presented (see Défago et al. [10] for a survey). The text in this section is largely inspired from [10]

### 2.2.1 Specification of Total Order Broadcast

The problem is defined in terms of two primitives, which are called *TO-broadcast(m)* and *TO-deliver(m)*, where $m$ is some message. When a process $p$ executes *TO-broadcast(m)* (respectively *TO-deliver(m)*), we say that $p$ TO-broadcasts $m$ (respectively TO-deliver $m$). We assume that every message $m$ can be uniquely identified, and carries the identity of its sender, denoted by *sender(m)*. In addition, we assume that, for any given message $m$, and any run, *TO-broadcast(m)* is executed at most once. In this context, total order broadcast is defined by the following properties (Hadzilacos and Toueg 1994; Chandra and Toueg 1996). [16, 7]:

- (VALIDITY). If a correct process TO-broadcasts a message $m$, then it eventually TO-delivers $m$.
- (UNIFORM AGREEMENT). If a process TO-delivers a message $m$, then all correct processes eventually TO-delivers $m$.
- (UNIFORM INTEGRITY). For any message $m$, every process TO-delivers $m$ at most once, and only if $m$ was previously TO-broadcast by *sender(m)*.
- (UNIFORM TOTAL ORDER). If a processes $p$ and $q$ both TO-deliver messages $m$ and $m'$, then $p$ TO-delivers $m$ before $m'$ if and only if $q$ TO-delivers $m$ before $m'$.

Validity and Uniform Agreement are liveness properties, while Uniform Integrity and Uniform Total Order are safety properties for the Total Order Broadcast.

### 2.2.2 Fault-tolerance issues

**Process Failures.** The specification of total order broadcast requires the definition of the notion of a *correct* process. The following set of process failure classes are commonly considered.

- Crash failures. When a process crashes, it ceases functioning forever. This means that it stops performing any activity including sending, transmitting, or receiving any message.
- Omission failures. When a process fails by omission, it omits performing some actions, such as sending or receiving a message.
- Timing failures. A timing failure occurs when a process violates some of the timing assumptions of the system model. Obviously, this type of failures does not exist in an asynchronous system models, because of the absence of timing assumptions in such systems.
- Byzantine failures. Byzantine failures are the most general type of failures. A Byzantine component is allowed any arbitrary behavior. For instance, a faulty process may change the content of messages, duplicate messages, send unsolicited messages, or even maliciously try to break down the whole system.

**Synchrony and timeliness** The synchrony of a system defines the timing assumptions that are made on the behavior of processes and communication channels. More specifically, one usually considers two major parameters. The first parameter is the *process speed interval*, which is given by the difference between the speed of the slowest and the fastest process in the system. The second parameter is the *communication delay*, which is given the time elapsed between the sending and the receipt of messages. The synchrony of the system is defined by considering various bounds on these two parameters.

A system where both parameters have a known upper bound is called a *synchronous system*. At the other extreme, a system in which process speed and communication delays are unbounded is called an *asynchronous system*.

There is an important theoretical result related to the consensus problem. It has been proven that there is no deterministic solution to the consensus problem in asynchronous distributed systems if just a single process can crash. [14].

Dolev et al. [11] showed that total order broadcast can be transformed into consensus, thus proving that the impossibility of consensus also holds for total order broadcast. These impossibility results were the motivation to extend the asynchronous system by introducing *oracles* to make consensus and total order broadcast deterministically solvable.

Chandra et al.[7] showed that consensus can be transformed to total order broadcast. The result holds also for arbitrary failures. Thus, consensus and total order broadcast are equivalent problems, that is, if there exists an algorithm that solves one problem, then it can be transformed into an algorithm that solves the other problem.

## 2.3 Failure detection

A recurrent pattern in all distributed algorithms is for a process $p$ to wait for a message from some other process $q$. If $q$ crashes, process $p$ is blocked. Failure detection is one basic mechanism to prevent $p$ from being blocked.

Unreliable failure detection has been formalized by (Chandra and Toueg 1996). [7] in terms of two properties: *accuracy* and *completeness*. Completeness prevents the blocking problem just mentioned. Accuracy prevents algorithms from running forever without solving the problem.

A failure detection is an oracle that provides information about the current status of processes, whether a given process has crashed or not. The notion of failure detection has been formalized by (Chandra and Toueg 1996). [7]. Briefly, a failure detector is modeled as a set of distributed modules, one module $FD_i$ is attached to each process $p_i$. Any process $p_i$ can query its failure detector module $FD_i$ about the status of other processes.

Failure detectors may be *unreliable* in the sense that they provide information that may not always correspond to the real state of the system. For instance a failure detector module $FD_i$ may provide the erroneous information that some process $p_j$ has crashed while, in reality, $p_j$ is correct and running. Conversely, $FD_i$ may provide the information that a process $p_k$ is correct while, $p_k$ has actually crashed.

To reflect the unreliability of the information provided by failure detectors, we say that a process $p_i$ *suspects* some process $P_j$ whenever $FD_i$ the failure detector module attached to $p_i$, returns the *unreliable* information that $p_j$ has crashed. In other words, a suspicion is a belief (e.g., "$p_i$ believes that $p_j$ has crashed") as opposed to a known fact (e.g., "$p_j$ has crashed and $p_i$ knows that").

There exists several classes of failure detectors, depending on how unreliable the information provided by the failure detector can be. Classes are defined by two properties, called *completeness* and *accuracy*, that constrain the range of possible mistakes.

We distinguish four classes of failure detectors, $\mathcal{P}$ (perfect), $\diamond\mathcal{P}$ (eventually perfect), $\mathcal{S}$ (strong), and $\diamond\mathcal{S}$ (eventually strong). The four classes share the same property of completeness, and only differs by their accuracy property. [7].

- STRONG COMPLETENESS Eventually every faulty process is permanently suspected by all correct processes.
- STRONG ACCURACY No process is suspected before it crashes. [class $\mathcal{P}$]
- EVENTUAL STRONG ACCURACY There is a time after which correct processes are not suspected by any correct process. [class $\diamond\mathcal{P}$]
- WEAK ACCURACY Some process is never suspected. [class $\mathcal{S}$]
- EVENTUAL WEAK ACCURACY There is a time after which some correct process is never suspected by any correct process. [class $\diamond\mathcal{S}$]

A failure detector of class $\diamond\mathcal{S}$ with a majority of correct processes allows to solve the *consensus* problem.[7]. Moreover, Chandra et al. [7] showed that a failure detector of class $\diamond\mathcal{S}$ is the weakest failure detector that allows to solve the consensus. The weakest failure detector to solve the consensus problem is said to be $\diamond\mathcal{W}$, which differs from $\diamond\mathcal{S}$ by satisfying a weak completeness property instead of Strong Completeness. However, Chandra et al. [7] proved the equivalence of $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$.

## 2.4 Consensus Problem

Consensus is defined by the primitives *propose(v)*, and *decide(v)*, which satisfy the following properties.

- TERMINATION. Every correct process eventually decides some value.
- UNIFORM INTEGRITY. Every process decides at most once.
- AGREEMENT. No two correct processes decide differently.
- UNIFORM VALIDITY. If a process decides $v$, then $v$ was proposed by some process.

Consensus can be solved in asynchronous systems prone to process crashes, augmented with failure detectors. In [7] Chandra and Toueg present two algorithms that solve consensus. One uses a failure detector of class $\mathcal{S}$ and tolerates $(n-1)$ faulty processes (in asynchronous systems with $n$ processes), and the other uses a failure detector of class $\diamond\mathcal{S}$ and tolerates ($f < n/2$) failures.

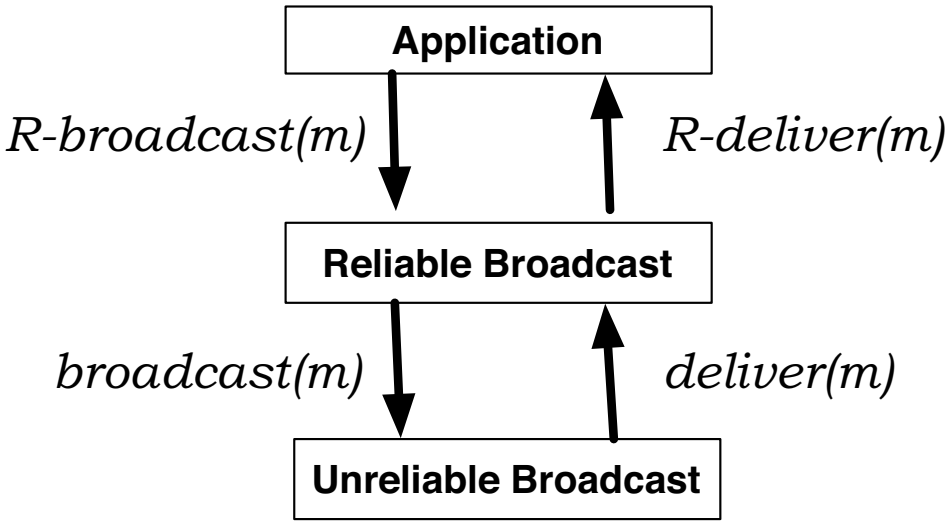



Figure 2.3: Reliable Broadcast primitives.

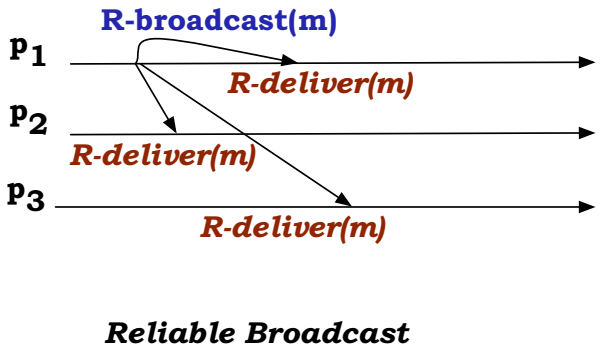



Figure 2.4: Reliable Broadcast agreement property.

Consensus and Total Order Broadcast have been shown in the literature to be equivalent in systems prone to process crashes. The equivalence result basically states that Total Order Broadcast can be reduced to Consensus, and Consensus can be reduced to Total Order Broadcast. The Consensus to Total Order Broadcast reduction consists in having *propose(v)* execute *TO-Broadcast*, and *decide(v)* occurring after the first *TO-Deliver(v)*.

## 2.5 Reliable Broadcast

Reliable Broadcast requires that all correct processes deliver the same set of messages (Agreement), and that this set includes all messages broadcast by correct processes (Validity), and no spurious messages (Integrity).

Formally, Reliable Broadcast is defined in terms of two primitives: R-broadcast($m$) and R-deliver($m$), which satisfy the following properties: [15]

Figure 2.3 illustrates the two primitives R-broadcast and R-deliver. Figure 2.4 and Figure 2.5

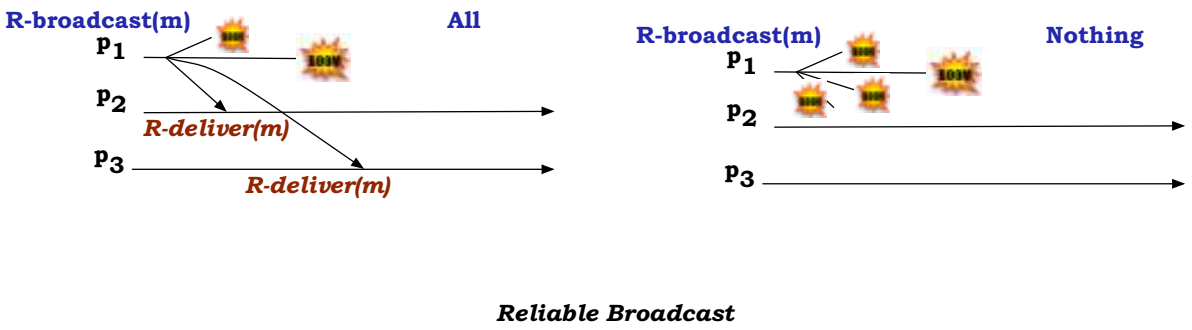



Figure 2.5: Reliable Broadcast (All or Nothing) property.

illustrates the properties of Reliable Broadcast.

- VALIDITY. If a correct process R-broadcasts a message *m* then, it eventually R-delivers *m*.
- AGREEMENT. If a correct process R-delivers a message *m* then, all correct processes eventually R-deliver *m*.
- UNIFORM INTEGRITY. For every message *m*, every process R-delivers *m* at most once, and only if *m* was previously R-broadcast by sender(*m*).

If a process *p* fails during the broadcast of a message, Reliable Broadcast allows two possible outcomes: either the message is delivered by *all* correct processes or by none.

Reliable Broadcast can be implemented by the following algorithm [7]. Whenever a process *p* R-broadcasts a message *m*, *p* sends *m* to all processes. Once a process *q* receives *m*, if $q \neq p$ then, *q* sends *m* to all processes, and in any case, *q* R-delivers *m*.

## 2.6 Group membership and view synchrony

The group membership is to maintain a list of currently active processes. The list can change with new members joining and old members leaving or crashing. Figure 2.6

Each process has a *view* of the list of currently active processes, and when this list changes, the *group membership* service reports the change to the processes by installing a new view. The group membership installs the same view at all correct processes.

A view *v* consists of a unique identifier and a list of processes (members of the view *v*). The group membership service maintains a list of currently active processes, in failure-prone distributed systems, and delivers this information to the application whenever its composition changes. (Figure 2.7 illustrates a group membership service for a distributed system in presence of failures).

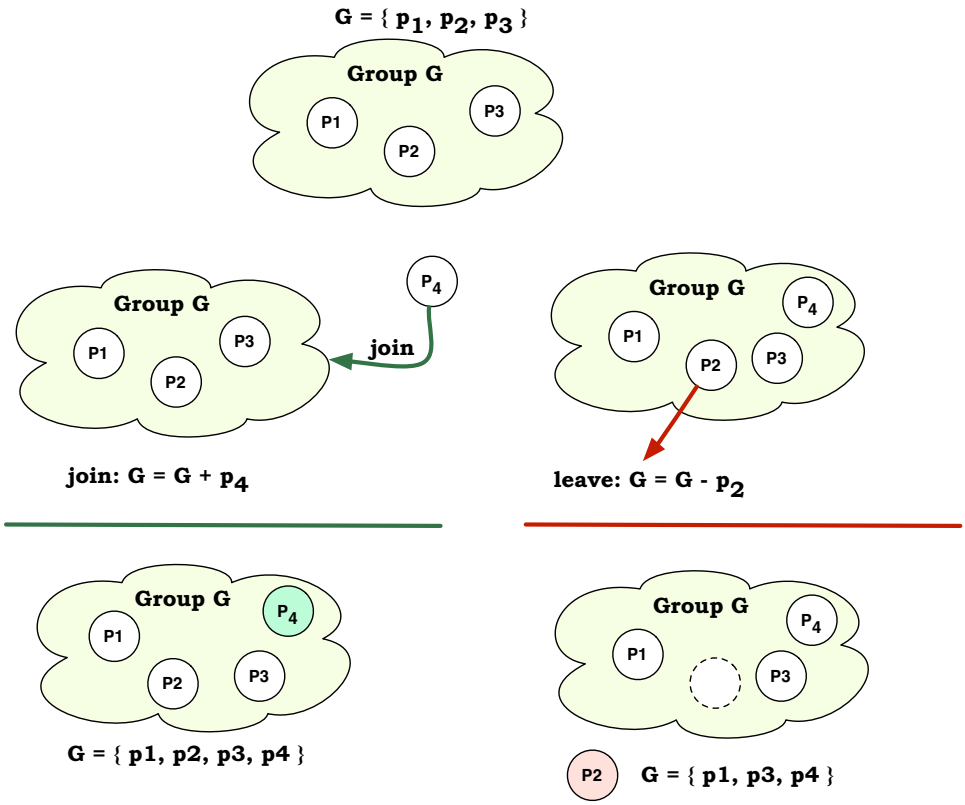



Figure 2.6: Group membership.

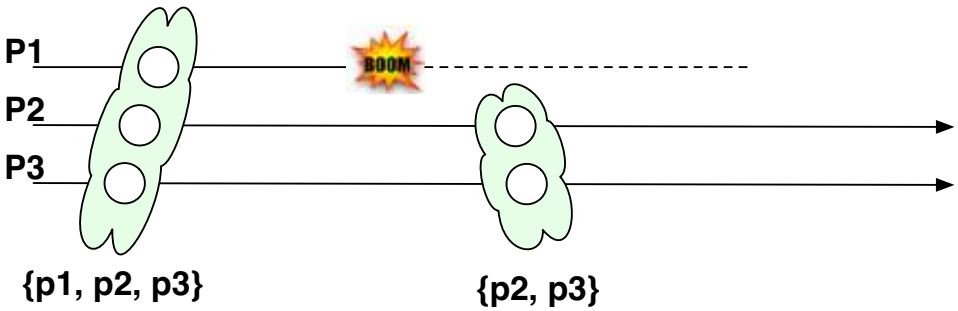



Figure 2.7: Group membership service in presence of failures.

The reliable multicast services deliver messages to the current view members. For more information on the subject, we refer to the survey of Chockler et al. [8]. A group membership can also be combined with failure detection [34], and then it can be seen as a high-level failure detection mechanism that provides consistent information about suspicions and failures [30, 17]. In short, a group membership keeps a track of what processes belong to the distributed computation and what processes does not.

A group membership service provides a list of non-crashed processes that currently belong to the system, and satisfies three properties [8]: validity, agreement and termination.

- Validity. The validity property is explained as follows: let $v_i$ and $v_{i+1}$ be two consecutive views, if a process $p \in v_i \setminus v_{i+1}$ then some process has executed *leave(p)* and if a process $p \in v_{i+1} \setminus v_i$ then some process has executed *join(p)*.

- Agreement. The agreement property ensures that the same view would be installed by all the processes of the group (agreement on the view) since agreement on uniquely identified views is necessary for synchronizing communications. So, if a process $p$ in view $v_i$ installs view $v_{i+1}$, and process $q$ in view $v_i$ installs view $v'_{i+1}$, then $v_{i+1} = v'_{i+1}$

- Termination. The termination property means that if a process $p \in v$ executes *join(q)*, then unless $p$ crashes, eventually a view $v'$ is installed such that either $q \in v'$ or $p \notin v'$.

# Chapter 3

# System models and definitions

## 3.1 System models

We consider a system of mobile robots $S = \{R_i\}$, in which each robot has a unique identifier.

Robots have access to a global positioning device that, when queried by a robot $R_i$, returns $R_i$'s position with a bounded error $\varepsilon_{gps}$. The robots communicate using wireless communication. Communications assume retransmissions mechanisms such that communication channels are reliable.

The system is asynchronous in the sense that there is no bound on communication delays, processing speed and on robots speed movement.

We consider two system models, the *closed group* and the *dynamic group* models.

**Closed group model** The closed group model consists of a static group of cooperative mobile robots, composed of $n$ mobile robots $S = \{R_1, \ldots, R_n\}$, such that the total composition of the group is known to each robot. A robot can always communicate with all robots of the group.

**Dynamic group model** The dynamic group model consists of a dynamic group of cooperative mobile robots. The entire composition of the system, of which robots have only a partial knowledge, can change dynamically.

Robots have limited communication range $D_{tr}$, hence they naturally form an ad hoc network on which they rely for their communication. If the distance between two robots $R_i$ and $R_j$ is less than $D_{tr}$, then the two robots can communicate with each other. Each robot has an access to a neighborhood discovery primitive named *NDiscover*, which is readily available through most of wireless communication devices.

## 3.2 Definitions

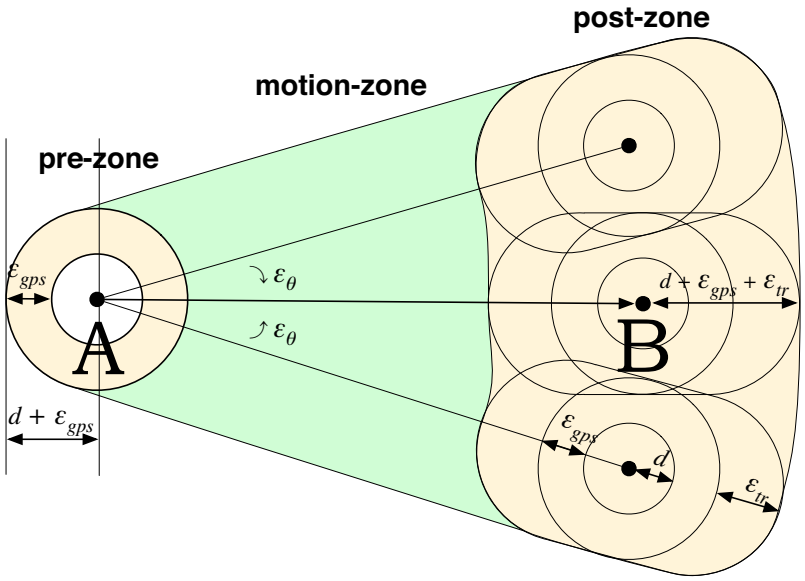



Figure 3.1: Reservation Zone.

A robotic application determines a destination that a robot has to reach. Then, the motion planning layer computes a path along which a robot moves to reach the goal. For convenience, a robot needs to move along a path by steps. Hence, a path determined by the motion planning layer, is divided into smaller parts, each of which is called a *chunk*.

**Paths.** We denote by *chunk* a line segment along which a robot moves. A path of a robot is a continuous route composed of a series of contiguous chunks. A path can take an arbitrary geometric shape, but we consider only line segment based paths for simplicity.

**Errors.** There are three sources of geometrical incertitude concerning the position and the motion of a robot. Error related to the position information provided by the positioning system denoted $\varepsilon_{gps}$. In addition, the motion of a robot creates two additional sources of errors, the first error is related to the translational movement, denoted: $\varepsilon_{tr}$. The second error is related to the rotational movement, denoted: $\varepsilon_\theta$.

**Zones.** A *zone* is defined as the area needed by a robot to move safely along a chunk. This includes provisions for the shape of the robot, positioning error and imprecisions in the moving of the robot. The zone must be a convex shape and contains the chunk the robot is following. Figure 3.1 shows the zone $Zone_i$ for a robot $R_i$ moving along a chunk $AB$, where $d$ represents the radius of the geometrical shape of $R_i$. The zone $Zone_i$ is composed of the following three parts, illustrated in Figure 3.1: the first part named *pre-motion* zone and denoted $pre(Zone_i)$, is the zone that robot $R_i$ possibly occupies while waiting (before moving). The second part

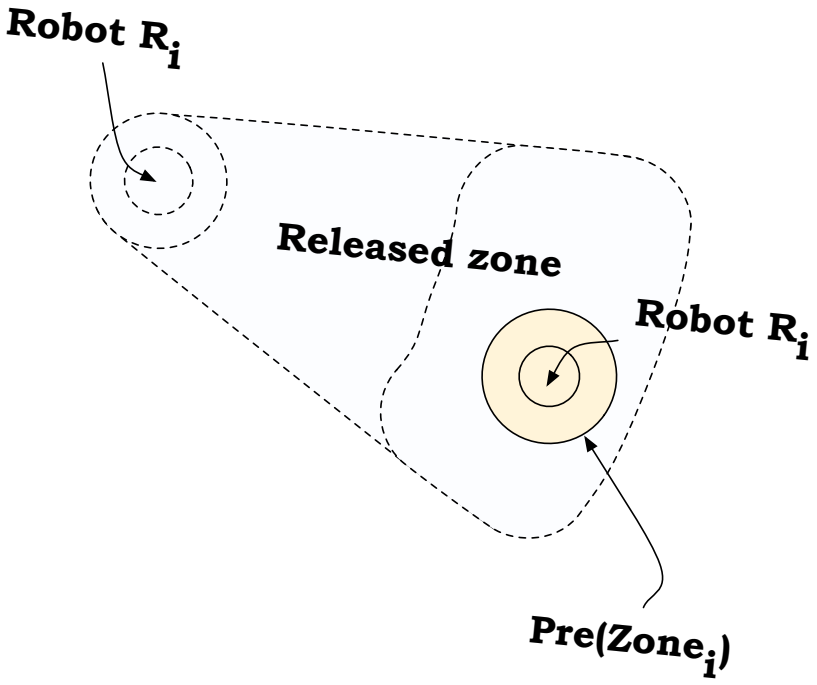



Figure 3.2: A robot $R_i$ releases the pervious zone and keeps only the place that may occupy $pre(Zone_i)$

named *motion* zone and denoted $motion(Zone_i)$, is the zone that robot $R_i$ possibly occupies while moving. The third part named *post-motion* zone and denoted $post(Zone_i)$, is the zone that robot $R_i$ possibly reaches after the motion.

We say that a robot $R_i$ is the *owner* of a zone $Zone_i$ ($Zone_i$ is *granted* to $R_i$), if $R_i$ reserves $Zone_i$ and did not release it yet. A robot $R_i$ *releases* the zone $Zone_i$ that it has owned and keeps only a part of $post(Zone_i)$ under its reservation. The part of the zone that has been released by $R_i$ is denoted: $RelZone_i$. Figure 3.2 shows that the *pre-motion* zone $pre(Zone_i)$ is entirely included within the previous *post-motion* zone, and presents also the current and the previous positions of $R_i$.

Figure 3.3 presents the previously released zone Previous($RelZone_i$) and the current requested zone $Zone_i$.

**Releasing a zone.** $RelZone_i$ is the zone that a robot $R_i$ releases when $R_i$ reaches the post-motion zone $post(Zone_i)$.

$RelZone_i = pre(Zone_i) \cup motion(Zone_i) \cup \text{SubPost}(Zone_i)$, where: $\text{SubPost}(Zone_i) \subset post(Zone_i)$

$pre(Zone_i) \subset \text{PREVIOUS}(post(Zone_i))$

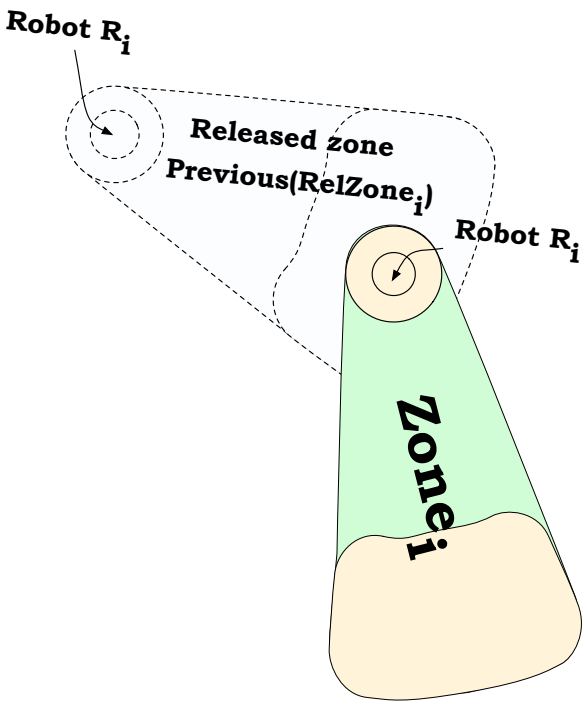



Figure 3.3: $Zone_i$ is the current requested zone by a robot $R_i$. Previous($RelZone_i$) is the previously released zone by $R_i$.

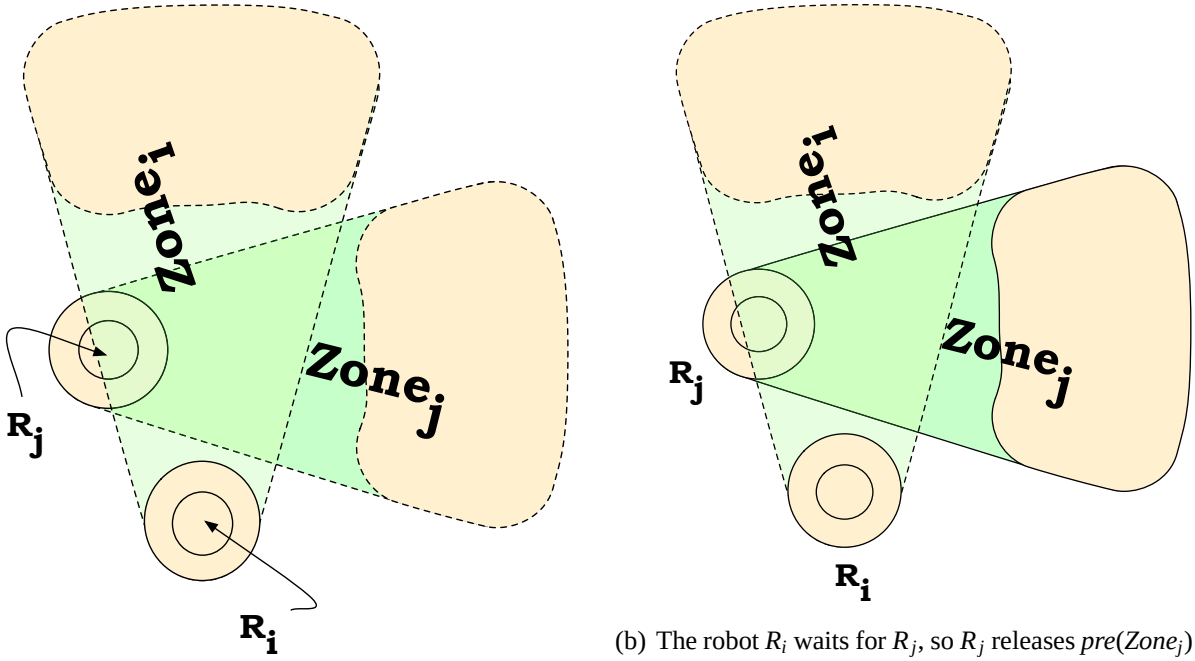
**Relation between robots.** We say that $R_i$ *conflict* with $R_j$ if the requested zone $Zone_i$ intersects with $Zone_j$ of robot $R_j$. If $R_i$ conflicts with $R_j$ then, one of them owns its requested zone and eventually releases it before the other robot owns its requested zone.

However, there are specified intersection situations between $Zone_i$ and $Zone_j$, such that neither $R_i$ nor $R_j$ can move. In this dissertation, we use the term "deadlock" to express specified intersection situations between a zone $Zone_i$ and a zone $Zone_j$. These intersection situations imply that neither $R_i$ can own $Zone_i$ nor $R_j$ can own $Zone_j$.

We say that a robot $R_i$ is in a deadlock situation with a robot $R_j$. For example, a deadlock situation between two robots, occurs when a robot requests a zone that intersects with both the *pre-motion* and the *post-motion* zones of the other, so none of the robots can move. The conditions and expressions of the deadlock situations are discussed in details in Subsection 3.2.1.

### 3.2.1 Deadlock situations

There are pathological intersection situations between $Zone_i$ and $Zone_j$, such that neither $R_i$ nor $R_j$ can move, or if one of them has granted its zone before the other then, the other robot may not be able to own its requested zone (starvation situation). We say that $R_i$ and $R_j$ are in a deadlock situation because none of the robots can own its requested zone, or if one of the robot owns its requested zone then, the other robot starves.




(a) Deadlock risk$_{pre}$($R_i$, $R_j$). $Zone_i$ intersects with $pre$($Zone_j$) and the *post- motion* zones do not intersect.

(b) The robot $R_i$ waits for $R_j$, so $R_j$ releases $pre$($Zone_j$) before $R_i$ owns $Zone_i$.

Figure 3.4: $Zone_i$ intersects with $pre$($Zone_j$) and the post-motion zones do not intersect.

A robot $R_i$ is in a *deadlock risk* situation with robot $R_j$, when the requested zone $Zone_i$ intersects either with the *pre-motion* or with the *post-motion* zone of the robot $R_j$.

This situation is called *deadlock risk* because a deadlock situation occurs if $Zone_i$ intersects with *both* $pre$($Zone_j$) and $post$($Zone_j$). A deadlock situation occurs also if $post$($Zone_i$) intersects with $post$($Zone_j$). The other possible deadlock situations between two robots occur when each robot requests a zone that intersects with the *pre-motion* zone of the other, and when the *motion* zone of each robot intersects with the *post-motion* zone of the other.

A robot $R_i$ is in a *deadlock risk*$_{pre}$ situation with robot $R_j$ if $Zone_i$ intersects with $pre$($Zone_j$) and the *post-motion* zones do not intersect. A deadlock risk$_{pre}$($R_i$, $R_j$) situation is presented in Figure 3.4(a).

A robot $R_i$ is in a *deadlock risk*$_{post}$ with $R_j$ if $motion$($Zone_i$) intersects with $post$($Zone_j$) and the *post-motion* zones do not intersect . A deadlock risk$_{post}$($R_i$, $R_j$) situation is presented in Figure 3.6(a).

- Deadlock risk$_{pre}$($R_i$, $R_j$): [$Zone_i \cap pre(Zone_j) \neq \emptyset$] and [$post(Zone_i) \cap post(Zone_j) = \emptyset$]
- Deadlock risk$_{post}$($R_i$, $R_j$): [$Zone_i \cap post(Zone_j) \neq \emptyset$] and [$post(Zone_i) \cap post(Zone_j) = \emptyset$]

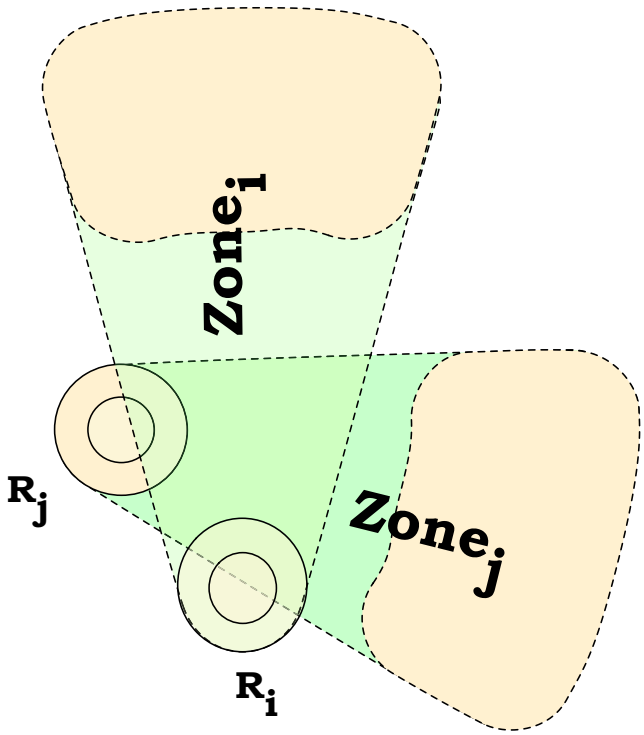
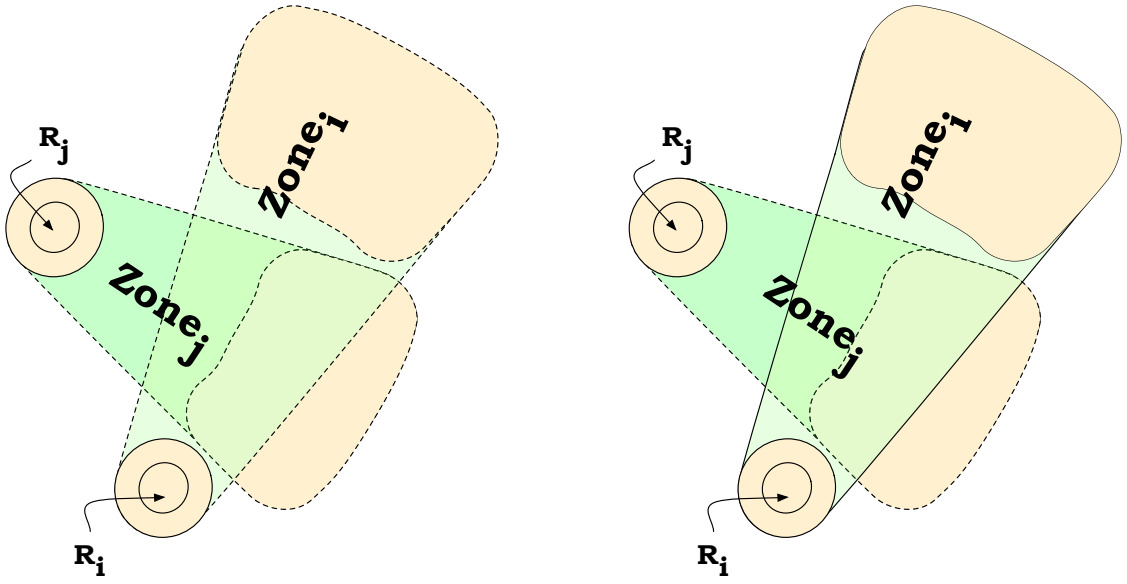



Figure 3.5: Deadlock situation 1: $Zone_i$ intersects with $pre(Zone_j)$ and $Zone_j$ intersects with $pre(Zone_i)$ and the post-motion zones do not intersect.




(a) Deadlock risk$_{post}(R_i, R_j)$. $Zone_i$ intersects with $post(Zone_j)$ and the *post- motion* zones do not intersect.

(b) The robot $R_j$ waits for $R_i$, so $R_i$ releases $motion(Zone_i)$ before $R_j$ owns $Zone_j$.

Figure 3.6: $Zone_i$ intersects with $post(Zone_j)$ and the post-motion zones do not intersect.

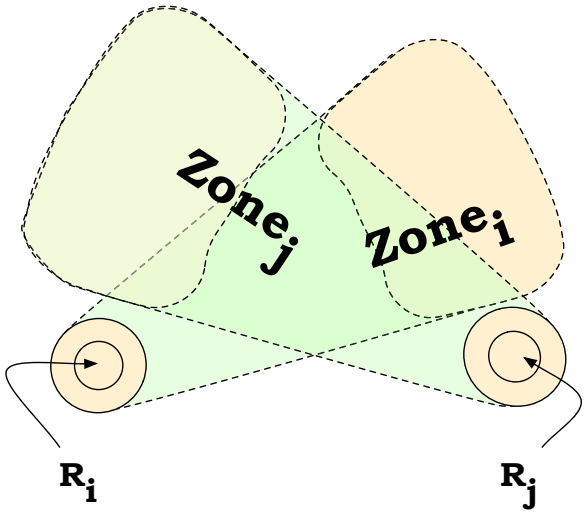



Figure 3.7: Deadlock situation 2: $Zone_i$ intersects with $post(Zone_j)$ and $Zone_j$ intersects with $post(Zone_i)$ and the post-motion zones do not intersect.

**Conditions and expressions of deadlock situations** The deadlock situations are as follows.

1. Deadlock situation 1: Deadlock risk$_{pre}(R_i, R_j)$ and Deadlock risk$_{pre}(R_j, R_i)$. The Deadlock situation 1 is illustrated in Figure 3.5.

2. Deadlock situation 2: Deadlock risk$_{post}(R_i, R_j)$ and Deadlock risk$_{post}(R_j, R_i)$. The Deadlock situation 2 is illustrated in Figure 3.7.

3. Deadlock situation 3: Deadlock risk$_{pre}(R_i, R_j)$ and Deadlock risk$_{post}(R_i, R_j)$. The Deadlock situation 3 is illustrated in Figure 3.8.

4. Deadlock situation 4: $post(Zone_i) \cap post(Zone_j) \neq \emptyset$. The Deadlock situation 4 is illustrated in Figure 3.9.

The deadlock situations are expressed as follows.

1. Deadlock situation 1:
   $[Zone_i \cap pre(Zone_j) \neq \emptyset]$ and $[Zone_j \cap pre(Zone_i) \neq \emptyset]$ and $[post(Zone_i) \cap post(Zone_j) = \emptyset]$

2. Deadlock situation 2:
   $[Zone_i \cap post(Zone_j) \neq \emptyset]$ and $[Zone_j \cap post(Zone_i) \neq \emptyset]$ and $[post(Zone_i) \cap post(Zone_j) = \emptyset]$

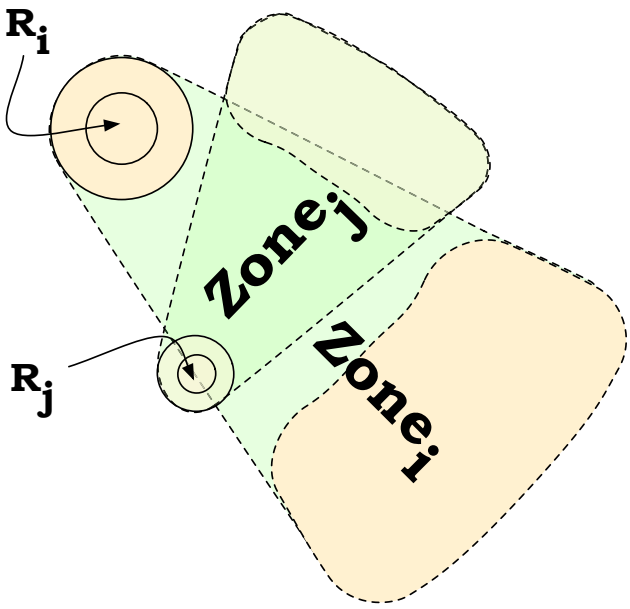



Figure 3.8: Deadlock situation 3: $Zone_i$ intersects with both ($pre(Zone_j)$ and $post(Zone_j)$).

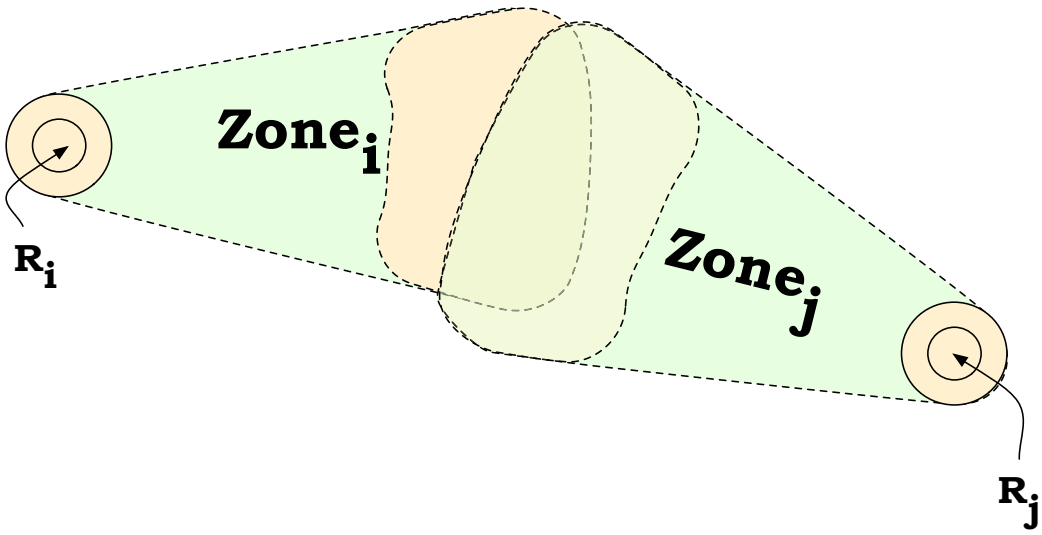



Figure 3.9: Deadlock situation 4: $post(Zone_i)$ intersects with $post(Zone_j)$.

3. Deadlock situation 3:
   $[Zone_i \cap pre(Zone_j) \neq \emptyset]$ and $[Zone_i \cap post(Zone_j) \neq \emptyset]$ and $[post(Zone_i) \cap post(Zone_j) = \emptyset]$

4. Deadlock situation 4:
   $post(Zone_i) \cap post(Zone_j) \neq \emptyset$

**Definition 1** *DS is the set of the four deadlock situations.*
$DS = \{Deadlock\ situation\ 1, Deadlock\ situation\ 2, Deadlock\ situation\ 3, Deadlock\ situation\ 4\}$.

**Imposed wait-for relations between robots** The *deadlock risk* situations between a robot $R_i$ and a robot $R_j$: deadlock $risk_{pre}(R_i, R_j)$ and deadlock $risk_{post}(R_i, R_j)$ imposes wait-for relations between $R_i$ and $R_j$.

- If $R_i$ is in *deadlock* $risk_{pre}$ situation with $R_j$, then $R_i$ must wait for $R_j$. So that, $R_j$ releases $pre(Zone_j)$ before $R_i$ owns $Zone_i$. Figure 3.4(b) illustrates that $R_i$ must wait for $R_j$.

- If $R_i$ is in *deadlock* $risk_{post}$ situation with $R_j$, then $R_j$ must wait for $R_i$. So that, $R_i$ releases $motion(Zone_i)$ before $R_j$ owns $Zone_j$. Figure 3.6(b) illustrates that $R_j$ must wait for $R_i$.

# Chapter 4

# Collision prevention problem: definition and specification

## 4.1 Collision Prevention: problem

The collision prevention consists of a distributed path reservation system, such that a robot must reserve a zone before it moves. When a robot reserves a zone, it can move *safely* inside the zone. The path reservation is performed in a consistent manner. All robots run the same protocol. When a robot wants to move along a given chunk, it must reserve the zone that surrounds this chunk. When this zone is reserved, the robot moves along the chunk. Once the robot reaches the end of the chunk, it releases the zone except for the area that the robot occupies. When moving along a path, the robot repeats this procedure for each chunk along the path.

## 4.2 Problem definition and specification

A robot can move only in a zone that it owns. When a robot $R_i$ requests a zone $Zone_i$, it invokes the primitive *req*. We say that $Zone_i$ is *granted* to $R_i$ ($R_i$ *owns* $Zone_i$) upon return from *req* and invocation of the primitive *reserve*. When $R_i$ *reserves* $Zone_i$, it moves along the chunk. Once the robot reaches the end of the chunk, it *releases* $Zone_i$ upon return from *reserve* and invocation of the primitive *release*. If a robot requests a zone, then either the robot is granted the zone or it receives an exception. The relationship between robots and zone changes in time. A zone is said to be free if it is not owned by any robot. In order to resolve the collision prevention problem, and to keep the system of mobile robots always in progress towards its final goal, certain properties of *safety* and *liveness* must hold. If a robot requests a zone, then eventually it owns this zone or receives an exception. We say that the robot owns the zone and all the points contained in this zone. A given point can be owned by only one robot. If a robot owns a zone, it eventually releases that zone.

**Property 1 (Mutual exclusion)** *If the requested zone $Zone_i$ of $R_i$ intersects with the requested zone $Zone_j$ of $R_j$ then exclusively either $R_i$ or $R_j$ becomes the owner of its requested zone.*

Consequently, a point in the plane can be owned by only one robot.

$(Zone_i \cap Zone_j \neq \emptyset) \Rightarrow (R_i \text{ owns } Zone_i)$ XOR $(R_j \text{ owns } Zone_j)$

**Property 2 (Liveness)** *If a robot $R_i$ requests $Zone_i$ then eventually ($R_i$ owns $Zone_i$ or an exception is raised).*

$R_i$ requests $Zone_i \Rightarrow \Diamond$ ($R_i$ owns $Zone_i$ or Exception)

**Property 3 (Non triviality)** *Exception is raised only if a deadlock situation occurs.*

The following property must hold to ensure the *integrity* of the system. If a robot owns a zone, then eventually it leaves that zone. If a robot leaves a zone, then it releases that zone.

# Chapter 5

# Collision prevention protocol for a closed group model

## 5.1 Closed group model

We consider a system of $n$ mobile robots $S = \{R_1, \ldots, R_n\}$, in which each robot has a unique identifier. The total composition of the system is known to each robot.

Robots have access to a global positioning device that, when queried by a robot $R_i$, returns $R_i$'s position with a bounded error $\varepsilon_{gps}$.

The robots communicate using wireless communication such that a robot $R_i$ can communicate with all robots of the system. Communications assume retransmissions mechanisms such that communication channels are reliable.

The system is asynchronous in the sense that there is no bound on communication delays, processing speed and on robots speed movement.

## 5.2 Collision Prevention: protocol

**Idea of the protocol.** All robots run the same distributed algorithm which is based on the following idea. When a robot $R_i$ requests a zone $Zone_i$, $R_i$ broadcasts a message indicating a request of a zone REQUEST($Zone_i$)and a release of the previous owned zone. RELEASE(PREVIOUS($RelZone_i$)).

A wait-for graph is generated according to the delivered requests and releases. The wait-for graph represents the wait-for relations between robots. If a requested zone $Zone_i$ of a robot $R_i$ intersects with a requested zone $Zone_j$ of robot $R_j$, then a wait-for relation between $R_i$ and $R_j$ is established. When a robot $R_i$ reaches the *post-motion* zone $post(Zone_i)$, $R_i$ releases the previous zone, and requests a new zone.

All the robots in the system deliver requests and releases in the same order, thus consistent reservations and releases of zones take place.

### 5.2.1 Variables

We present the variables used in the protocol.

- $Zone_i$ is the zone currently requested or owned by robot $R_i$.
- DELIVERED is an ordered set: $\{(R_i, \text{REQUEST}, Zone_i, \text{RELEASE}, \text{PREVIOUS}(RelZone_i))\}$. The list DELIVERED represents the TO-delivery of the Total Order Broadcast algorithm. A robot $R_i$ TO-broadcasts a request of a zone $Zone_i$ and a release of the previous zone PREVIOUS($RelZone_i$). The Total Order Broadcast ensures that all robots in the system deliver the same list DELIVERED.
- $Dag_{arbiter}$ is a directed acyclic graph generated by the Arbiter algorithm. $Dag_{arbiter}$ is a wait-for graph such that the vertices represent the robots whose requests belong to the list DELIVERED, and a directed edge from vertex($R_i$) to vertex($R_j$) indicates that $R_i$ waits for $R_j$.
- $Dag_{wait}$ is a directed acyclic graph that is generated by appending the Arbiter graphs $Dag_{arbiter}$ generated by successive batches of the protocol.

### 5.2.2 Protocol description

We explain the phases of the protocol with respect to a robot $R_i$. The robot $R_i$ is located in the *pre-motion* zone $pre(Zone_i)$. When robot $R_i$ requests a new zone $Zone_i$, it proceeds as follows.

1. TO-broadcast:

   $R_i$ performs a total order broadcast of a message carrying a REQUEST with the parameters of the requested zone $Zone_i$, and also a RELEASE with the parameters of the released zone PREVIOUS($RelZone_i$). The robot $R_i$ releases the previous zone and requests a new zone $Zone_i$.

2. TO-deliver:

   The TO-deliver of the total order broadcast algorithm returns an ordered set denoted by DELIVERED which is composed of requested and released zones. DELIVERED corresponds to a batch of the protocol. The total order broadcast guarantees that the list DELIVERED of $R_i$ is identical (with respect to the composition and to the order of the elements) to the list DELIVERED of a robot $R_j$ of the group.

3. Arbiter:

All robots deliver the same set of requested and released zones in the same order. However, the collision prevention problem imposes a wait-for relation between two requested zones which might be different from the order delivered by the total order broadcast.

In addition, the sequential order of requests delivered by the total order broadcast imposes that a robot $R_i$ must wait for all the robots that their request's index in DELIVERED is smaller than that of $R_i$'s request, even if the requested zones do not intersect.

The Arbiter algorithm generates a directed acyclic graph based on the list DELIVERED such that the requested zones that do not intersect with other zones are granted to the corresponding robots simultaneously, and a robot $R_i$ directly waits only for robots $R_j$ if $Zone_j$ intersects with $Zone_i$.

The Arbiter module starts by checking the deadlock situations between the requested zones of the list DELIVERED. (The deadlock situations are illustrated in Chapter 3). If a deadlock situation is detected, then the Deadlock Handler is called to resolve the deadlock situation.

The algorithm Arbiter generates an acyclic directed graph $Dag_{arbiter}$ as follows. The vertices of the graph $Dag_{arbiter}$ represent robots and a directed edge between two vertices represents a wait-for relation between the corresponding robots. The Arbiter algorithm scans the list DELIVERED according to the order delivered by the total order broadcast. The algorithm compares each requested zone of a robot $R_x$ with all the requested zones of robots $R_y$ of larger request's index in the list DELIVERED. If $Zone_x$ intersects with $Zone_y$ then $R_y$ waits for $R_x$ because the index of $R_y$ is larger than the index of $R_x$ in the list DELIVERED. A directed edge from the vertex of $R_y$ to the vertex of $R_x$ is added to the graph $Dag_{arbiter}$.

**Imposed wait-for relations**

- If $[Zone_i \cap pre(Zone_j) \neq \emptyset]$ and $[post(Zone_i) \cap post(Zone_j) = \emptyset$ then, $R_i$ must wait for $R_j$.
- If $[Zone_i \cap post(Zone_j) \neq \emptyset]$ and $[post(Zone_i) \cap post(Zone_j) = \emptyset]$ then, $R_j$ must wait for $R_i$.

The Arbiter algorithm performs deterministic computations based on the list DELIVERED such that all the requesting robots generates the *same* directed acyclic graph $Dag_{arbiter}$.

4. Append-Graphs:

   The algorithm Append-Graphs appends the generated graph $Dag_{arbiter}$ to the wait-for graph $Dag_{wait}$ using the list DELIVERED. At first, the algorithm updates the graph $Dag_{wait}$

by removing the vertices that correspond to the released zones of the list DELIVERED. After that, the algorithm *appends* $Dag_{arbiter}$ the wait-for graph $Dag_{wait}$ such that a robot $R_i$ that belongs to $Dag_{arbiter}$ must wait-for a robot $R_j$ that belongs to $Dag_{wait}$ if $Zone_i$ intersects with $Zone_j$.

**A deadlock situation** If the requested zone $Zone_i$ of $R_i$ in $Dag_{arbiter}$ intersects with the *post-motion* zone of $R_j$ in $Dag_{wait}$, then a deadlock occurs because $R_i$ must causally wait for $R_j$, however $Zone_i$ intersects with $post(Zone_j)$. The Deadlock handler is responsible for resolving this problem, which can be done by preempting the request of $R_i$ and then $R_i$ requests an alternative zone.

5. Reservation: When all the robots that $R_i$ is waiting for, release their zones, the request procedure of $R_i$ returns and $R_i$ reserves $Zone_i$, hence $R_i$ becomes the owner of $Zone_i$.

6. Release: When $R_i$ reaches the *post-motion* zone $post(Zone_i)$, it computes its new position and thus it computes the zone to be released which is $Zone_i$ except the place that $R_i$ may possibly occupy (footprint and the positioning system error $\varepsilon_{gps}$). $R_i$ performs a total order broadcast of a message carrying the parameters of the next requested zone and the parameters of the current released zone. Initially, the released zone is set to $\perp$. If a robot does not acquire a next zone, then $Zone_i$ is set to $\perp$.

   The total order broadcast of a message carrying both the next requested zone and the current released zone guarantees that all the robots deliver the same list DELIVERED and hence generates the same graph $Dag_{arbiter}$ which enables to update the wait-for graph consistently, by removing the vertices corresponding to the robots that has released their zones, and by appending the graph $Dag_{arbiter}$ which represents the wait-for relations between the robots that have requested the next zones.

### 5.2.3 Deadlock Handler

When a deadlock situation is detected, then the Deadlock Handler is called to resolve the deadlock situation. The policy used by the Deadlock Handler to resolve deadlock situations is based on a *Request Preemption* strategy, which is summarized by preempting the request of the robot that has the larger index in the list DELIVERED. Then, the robot that has the larger index, restarts a request of an alternative zone.

The design of the collision prevention protocol yields a flexibility to handle the exceptions caused by deadlock situations, due to the module Deadlock Handler. The Deadlock Handler can apply a policy that is an application-dependent policy, in order to resolve deadlock situations.

**Algorithm 1** Collision prevention protocol (Code for robot $R_i$)

1: Initialisation:
2: PREVIOUS($RelZone_i$) := $\perp$; $Dag_{wait}$ := $\perp$; DELIVERED := $\emptyset$;

3: **procedure** Request($Zone_i$)
4: *TO-broadcast* [REQUEST, $Zone_i$, RELEASE, PREVIOUS($RelZone_i$)] {$R_i$ *TO-broadcasts a request of a new zone* $Zone_i$ *and a release of* PREVIOUS*(*$RelZone_i$*)*}
{$Zone_i$ *is set to* $\perp$ *if* $R_i$ *does not acquire to move any more*}

5: **when** *TO-Deliver* [REQUEST, $Zone_j$, RELEASE, PREVIOUS($RelZone_j$)]
6: DELIVERED := DELIVERED $\cup$ (REQUEST, $Zone_j$, RELEASE, PREVIOUS($RelZone_j$))
7: $Dag_{arbiter}$ := Arbiter(DELIVERED) {*Apply the deterministic function Arbiter(*DELIVERED*) to decide the wait-for relations*}
8: $Dag_{wait}$ := Append-Graphs($Dag_{wait}$, $Dag_{arbiter}$, DELIVERED) {*Append algorithm*}
{*update* $Dag_{wait}$ *by removing the released requests and appending the new requests*}

9: **when** the vertex of $R_i$ in $Dag_{wait}$ becomes a *sink* vertex (has no outgoing edges)
10: **return** {*all robots* $R_j$ *that* $R_i$ *waits for, has released their zones*}
11: **end when**
12: **end when**
13: **end** Request($Zone_i$)

**Algorithm 2** Arbiter

1: **function** Arbiter(*list*)
2: Deadlock-detector(*list*) {*If a deadlock situation then, the Deadlock detector calls the Deadlock Handler*}

3: **for** each index $x$ from MININDEX(*list*) to MAXINDEX(*list*) such that $R_x \in list$ **do**
4: **for** each index $y > x$ to MAXINDEX(*list*) such that $R_y \in list$ **do**
5: **if** Deadlock risk$_{pre}$($R_x$, $R_y$) or Deadlock risk$_{post}$($R_y$, $R_x$) **then**
6: $Dag_{arbiter}$ := $Dag_{arbiter}$ $\cup$ DirEdge($R_x$, $R_y$) {$R_x$ *must wait-for* $R_y$}
7: **end if**
8: **if** Deadlock risk$_{post}$($R_x$, $R_y$) or Deadlock risk$_{pre}$($R_y$, $R_x$) **then**
9: $Dag_{arbiter}$ := $Dag_{arbiter}$ $\cup$ DirEdge($R_y$, $R_x$) {$R_y$ *must wait-for* $R_x$}
10: **end if**
11: **if** Conflict($R_x$, $R_y$) and no edge ($R_x$, $R_y$) **then**
12: $Dag_{arbiter}$ := $Dag_{arbiter}$ $\cup$ DirEdge($R_y$, $R_x$) {$x, y$ *represent the indexes of robots* $R_x$ , $R_y$ *in list and* $y > x$}
{*a directed edge from a vertex of higher index in list to a vertex of lower index*}
{*The wait-for relation between two robots is set according to their indexes in list*}
13: **end if**
14: **end for**
15: **end for**
16: **return** $Dag_{arbiter}$
17: **end**

**Algorithm 3** Append-Graphs

1: **function** Append-Graphs($Dag_{wait}$, $Dag_{arbiter}$, *list*)
2: **for** all (RELEASE, PREVIOUS($RelZone_j$) $\neq \perp$) $\in$ *list* **do**
3: remove the vertex representing $R_j$ and its incoming edges from $Dag_{wait}$ {*Update the current wait-for graph* $Dag_{wait}$, *since* $R_j$ *releases the zone* PREVIOUS*(*$RelZone_j$*)*}
4: **end for**

5: **for** all $R_y \in Dag_{arbiter}$ **do**
6: **for** all $R_x \in Dag_{wait}$ **do**
7: **if** $Zone_y$ intersects with $Zone_x$ **then**
8: **if** $Zone_y$ intersects with $post(Zone_x)$ **then**
9: Deadlock-Handler($R_x$, $Zone_x$, $R_y$, $Zone_y$) {$R_y$ *must wait-for* $R_x$, *but if* $Zone_y$ *intersects with* $post(Zone_x)$ *then* $R_y$ *calls the Deadlock-Handler*}
10: **else**
11: $Dag_{wait} := Dag_{wait} \cup$ DirEdge($R_y$, $R_x$) {$R_y$ *must wait-for* $R_x$}
12: **end if**
13: **end if**
14: **end for**
15: **end for**
16: **return** $Dag_{wait}$
17: **end**

**Algorithm 4** Deadlock detector algorithm

1: **function** Deadlock-detector (*list*)
2: **if** $Zone_i \in$ *list* intersects with $pre(Zone_j)$ of $R_j$ that does not have a request in *list* **then**
3: Deadlock Handler($R_x$, $Zone_x$, $R_y$, $Zone_y$)
4: **end if**

5: **for** all ($R_x$, $R_y$) $\in$ *list* **do**
6: Deadlock situation 1:=
[$Zone_x \cap pre(Zone_y) \neq \emptyset$] and [$Zone_y \cap pre(Zone_x) \neq \emptyset$] and [$post(Zone_x) \cap post(Zone_y) = \emptyset$]
7: Deadlock situation 2:=
[$Zone_x \cap post(Zone_y) \neq \emptyset$] and [$Zone_y \cap post(Zone_x) \neq \emptyset$] and [$post(Zone_x) \cap post(Zone_y) = \emptyset$]
8: Deadlock situation 3:=
[$Zone_x \cap pre(Zone_y) \neq \emptyset$] and [$Zone_x \cap post(Zone_y) \neq \emptyset$] and [$post(Zone_x) \cap post(Zone_y) = \emptyset$]
9: Deadlock situation 4:=
$post(Zone_x) \cap post(Zone_y) \neq \emptyset$

10: **if** Deadlock situation 1 or Deadlock situation 2 or Deadlock situation 3 or Deadlock situation 4 **then**
11: Deadlock Handler($R_x$, $Zone_x$, $R_y$, $Zone_y$)
12: **end if**
13: **end for**
14: **end**

**Algorithm 5** Deadlock Handler algorithm

```
1:  function Deadlock-Handler (R_x, Zone_x, R_y, Zone_y)

2:     Event 1 := Zone_x intersects with pre(Zone_y) and R_y does not request a zone
3:     Event 2 := Zone_x intersects with post(Zone_y) and R_x must wait for R_y

4:     if Event 1 or Event 2 or Deadlock situation 3 then
5:        Request Preemption (R_x, Zone_x)
6:     else
7:        Request Preemption (the request with the higher index in DELIVERED)   {Deadlock situation 1 or
          Deadlock situation 2 or Deadlock situation 4}
8:     end if

9:     if no possible alternative chunk then
10:       throw Exception                                                       {There is no solution}
11:       return Exception
12:    end if
13:    Zone := Zone^alternative
14:    Request (Zone)                                   {Deadlock Handler proposes an alternative chunk}
15: end
```

If the Deadlock Handler does not find a solution, because there is no available alternative chunk then, the Deadlock Handler raises an exception.

### 5.2.4 Example.

Consider an application composed of the following six robots ($R_i$, $R_j$, $R_k$, $R_p$, $R_q$, $R_s$).

**First batch** The different intersections between the requested zones are represented in Figure. 5.1.

$Zone_i$ intersects with ($Zone_k$, $Zone_s$), $Zone_j$ intersects with $Zone_s$, $Zone_k$ intersects with ($Zone_i$, $Zone_s$), and $Zone_p$ does not intersect with any other requested zone.

- Each robot performs a total order broadcast of a message carrying the parameters of the requested zones. Each robot TO-deliver the same list DELIVERED.

  DELIVERED = [($R_k$, $Zone_k$), ($R_q$, $Zone_q$), ($R_s$, $Zone_s$), ($R_i$, $Zone_i$), ($R_p$, $Zone_p$), ($R_j$, $Zone_j$)].

- Generating the graph Arbiter. We assume in this example that there is no deadlock situations between the requested zones. Figure. 5.2 represents the generated directed acyclic graph $Dag_{arbiter}$.

  The graph Arbiter $Dag_{arbiter}$ is generated as follows. The algorithm checks the intersection of $Zone_k$ with the other requested zones in DELIVERED. $Zone_k$ intersects with both $Zone_i$

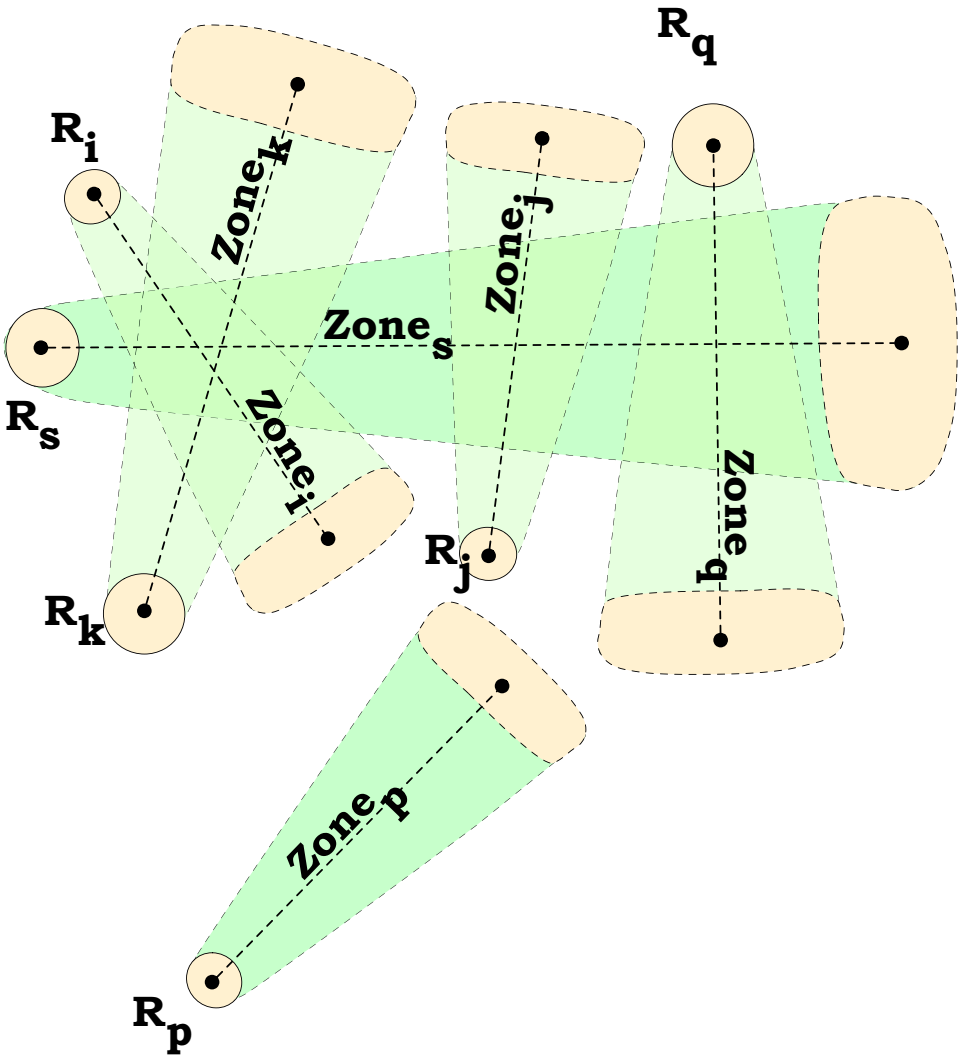



Figure 5.1: A group composed of six robots.

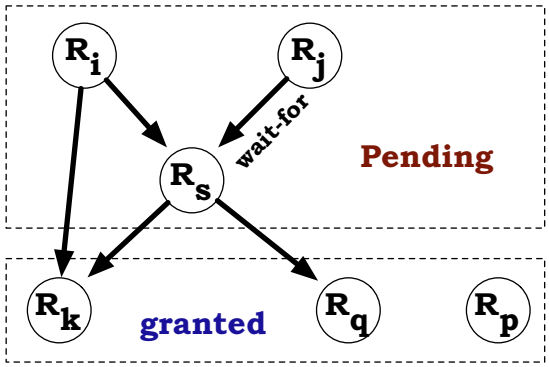



Figure 5.2: The directed acyclic graph generated by the Arbiter algorithm in the first batch.

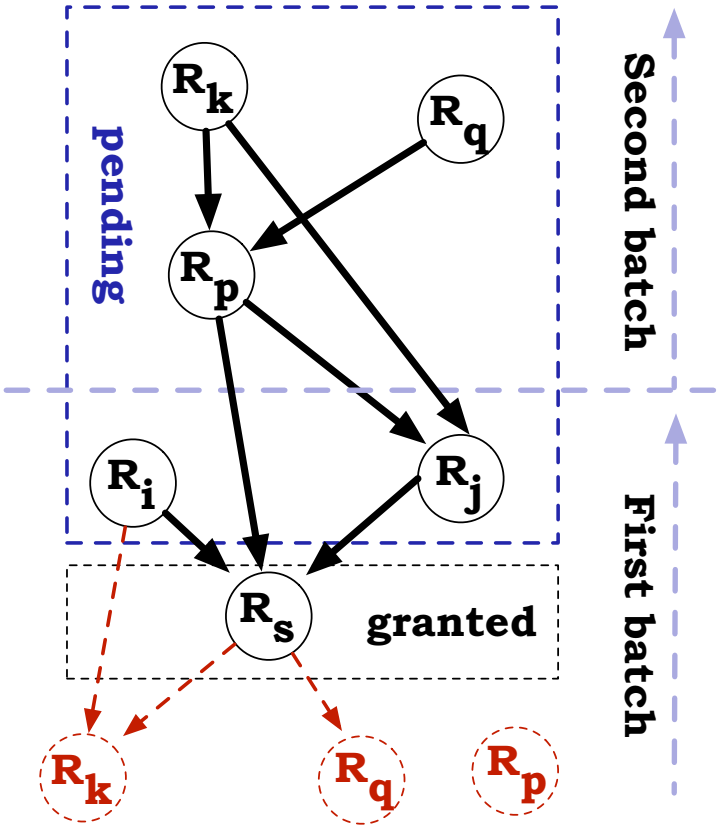



Figure 5.3: The resulting wait-for graph in the second batch.

and $Zone_s$. Since the indexes of requests in the list DELIVERED are ordered as follows: index($R_k$) < index($R_s$) < index($R_i$) then the directed edges between vertices representing robots are added to $Dag_{arbiter}$ as follows. A directed edge from the vertex($R_i$) to the vertex($R_s$), a directed edge from the vertex($R_s$) to the vertex($R_k$), and a directed edge from the vertex($R_i$) to the vertex($R_k$).

Then, the Arbiter algorithm checks the intersection of $Zone_q$ with the other requested zones, and adds a directed edge from the vertex($R_s$) to the vertex($R_q$). The graph $Dag_{arbiter}$ is presented in Figure. 5.2.

- The wait-for graph $Dag_{wait}$ is set to the graph $Dag_{arbiter}$ generated by the first batch of the protocol.

- In the wait-for graph $Dag_{wait}$ the vertices ($R_k$, $R_q$, $R_p$) are *sink* vertices (has no outgoing edges), so they do not wait for any robot. Therefore, they reserve the corresponding zones, and become the owners of ($Zone_k$, $Zone_q$, $Zone_p$) respectively.

**Second batch** Let us consider that ($R_k$, $R_q$, $R_p$) have reached the *post-motion* zones (*post*($Zone_k$), *post*($Zone_q$), *post*($Zone_p$)) respectively. Each of the robots $R_k$, $R_q$ and $R_p$ performs a total or-

der broadcast of a message carrying a request for a next zone $Zone_k$ and a release of the zone PREVIOUS($RelZone_k$). The intersections of the requested zones are as follows. $Zone_p$ intersects with both ($Zone_k$, $Zone_q$) but $Zone_k$ does not intersect with $Zone_q$. The second batch proceeds as follows.

- All robots deliver the list DELIVERED = [($R_p$, $Zone_p$), ($R_q$, $Zone_q$), ($R_k$, $Zone_k$)].
- Update the wait-for graph by removing the following vertices: vertex($R_p$), vertex($R_q$), and vertex($R_k$) in addition to their incoming edges, and append the graph $Dag_{arbiter}$.

  We assume that each of the following zones ($Zone_p$, $Zone_q$, $Zone_k$) do not intersect with a *post-motion* zone ($post(Zone_s)$, $post(Zone_i)$, $post(Zone_j)$).
- Append-Graphs. In order to append the graph $Dag_{arbiter}$ to the graph $Dag_{wait}$, the append algorithm (Algorithm. 3) checks the intersection between a zone that corresponds to a vertex of $Dag_{arbiter}$, and a zone that corresponds to a vertex of $Dag_{wait}$, for all vertices of both graphs. In this example, let us consider that $Zone_p$ intersects with $Zone_j$, then $R_p$ must wait for $R_j$. The graph $Dag_{arbiter}$ is appended to the graph $Dag_{wait}$ by adding a directed edge from the vertex($R_p$) to the vertex($R_j$). Figure. 5.3 represents the resulting wait-for graph $Dag_{wait}$ in the second batch.

  If there is no intersection between a zone corresponds to a vertex of the graph $Dag_{arbiter}$ and a zone corresponds to a vertex of the graph $Dag_{wait}$, then the resulting wait-for graph is composed of two disjoint subgraphs, thus there is no wait-for relation between a robot from one subgraph, and a robot from the other subgraph.

The following batches of the protocol take place exactly as explained in the second batch.

### 5.2.5 Proof of correctness

**Lemma 1** *If Deadlock* $risk_{pre}(R_i, R_j)$*, then* $R_i$ *must wait for* $R_j$.

**Proof.** The rule $R_i$ waits for $R_j \Rightarrow R_j$ releases $RelZone_j$ before $R_i$ owns $Zone_i$. Deadlock $risk_{pre}(R_i, R_j) \Rightarrow post(Zone_i)$ does not intersect with $post(Zone_j)$.

The released zone property shows that the *pre-motion* and the *motion* zone are included within the released zone (see Chapter 3).

Therefore, $Zone_i$ does not intersect with $Zone_j$, and the rule $R_i$ must wait for $R_j$ does not cause a collision between $R_i$ and $R_j$. □

**Lemma 2** *If Deadlock* $risk_{post}(R_i, R_j)$ *then* $R_j$ *must wait for* $R_i$.

**Proof.** The rule $R_j$ waits for $R_i \Rightarrow R_i$ releases $RelZone_i$ before $R_j$ owns $Zone_j$. Deadlock $\text{risk}_{post}(R_i, R_j) \Rightarrow post(Zone_i)$ does not intersect with $post(Zone_j)$.

The released zone property shows that $pre(Zone_i)$ and $motion(Zone_i)$ are included within the released zone $RelZone_i$ (see Chapter 3).

Therefore, $Zone_i$ does not intersect with $Zone_j$, and the rule $R_j$ must wait for $R_i$ does not cause a collision between $R_i$ and $R_j$. □

**Lemma 3** *The possible deadlock situations are as follows. A situation that belongs to the set* $DS = \{$*Deadlock situation 1, Deadlock situation 2, Deadlock situation 3, Deadlock situation 4*$\}$*, a situation where a requested zone intersects with the pre-motion zone of a robot that has not requested a zone, and a situation where a robot* $R_i$ *must wait for a robot* $R_j$*, and* $Zone_i$ *intersects with* $post(Zone_j)$.

**Proof.** At first, We prove that each of the situations of the set $DS$ is a deadlock situation.

**Deadlock situation 1**

- $R_i$ owns $pre(Zone_i)$. If $R_i$ waits for $R_j$, then $R_j$ owns $Zone_j$ and $R_i$ has owned $pre(Zone_i)$. The situation Deadlock $\text{risk}_{pre}(R_j, R_i)$ implies that $Zone_j$ intersects with $pre(Zone_i)$. Thus, a collision may occur between $R_i$ and $R_j$.

- $R_j$ owns $pre(Zone_j)$. If $R_j$ waits for $R_i$, then $R_i$ owns $Zone_i$ and $R_j$ has owned $pre(Zone_j)$. The situation Deadlock $\text{risk}_{pre}(R_i, R_j)$ implies that $Zone_i$ intersects with $pre(Zone_j)$. Thus, a collision may occur between $R_i$ and $R_j$.

So, neither $R_i$ nor $R_j$ can own the requested zone, ⇒ Deadlock situation 1 is a deadlock situation.

**Deadlock situation 2**

- If $R_i$ waits for $R_j$, then $R_j$ releases $RelZone_j$ before $R_i$ owns $Zone_i$. So, $R_j$ still owns $post(Zone_j) \subset SubPost(Zone_j)$. The situation Deadlock $\text{risk}_{post}(R_i, R_j)$ implies that $Zone_i$ intersects with $post(Zone_j)$. Thus a collision may occur between $R_i$ and $R_j$.

- If $R_j$ waits for $R_i$, then $R_i$ releases $RelZone_i$ before $R_j$ owns $Zone_j$. So, $R_i$ still owns $post(Zone_i) \subset SubPost(Zone_i)$. The situation Deadlock $\text{risk}_{post}(R_j, R_i)$ implies that $Zone_j$ intersects with $post(Zone_i)$. Thus, a collision may occur between $R_i$ and $R_j$.

So, neither $R_i$ nor $R_j$ can own the requested zone, ⇒ Deadlock situation 2 is a deadlock situation.

**Deadlock situation 3**

- If $R_i$ waits for $R_j$, then the situation Deadlock risk$_{post}$($R_i$, $R_j$) may cause a collision between $R_i$ and $R_j$ (previous arguments).
- If $R_j$ waits for $R_i$, then the situation Deadlock risk$_{pre}$($R_i$, $R_j$) may cause a collision between $R_i$ and $R_j$.

So, neither $R_i$ nor $R_j$ can own the requested zone, $\Rightarrow$ Deadlock situation 3 is a deadlock situation.

**Deadlock situation 4**

- If $R_i$ waits for $R_j$, then $R_j$ releases $RelZone_j$ before $R_i$ owns $Zone_i$, but $RelZone_j$ contains a part of $post(Zone_j)$ denoted by $SubPost(Zone_j)$ (see chapter 3). The part ($post(Zone_j) \subset SubPost(Zone_j)$) is still owned by $R_j$. When $R_i$ owns $Zone_i$ then it owns $post(Zone_i)$. Thus, a possible collision may occur between $R_i$ and $R_j$, since the *post-motion* zones $post(Zone_i)$ and $post(Zone_j)$ intersect.
- If $R_j$ waits for $R_i$ then a possible collision may occur between the two robots. (previous arguments)

So, neither $R_i$ nor $R_j$ can own the requested zone, $\Rightarrow$ Deadlock situation 4 is a deadlock situation. Consequently, the elements of $DS$ are deadlock situations.

We prove that a deadlock situation is one of the situations mentioned above in (Lemma 3).

A zone $Zone_i$ is composed of three zones: *pre-motion*, *motion*, and *post-motion* zone. So, $Zone_i = pre(Zone_i) \cup motion(Zone_i) \cup post(Zone_i)$. The possible intersection situations between two zones $Zone_i$ and $Zone_j$ are analyzed as follows.

- $pre(Zone_i) \cap pre(Zone_j) = \emptyset$. The intersection between $pre(Zone_i)$ and $pre(Zone_j)$ is impossible. $pre(Zone_i)$ is the zone that $R_i$ may occupy (footprint and $\varepsilon_{gps}$). The *pre-motion* zones are supposed not intersecting initially. During the run of the protocol, the *pre-motion* zone $pre(Zone_i)$ is entirely included within the previous *post-motion* zone. $pre(Zone_i) \subset$ PREVIOUS($post(Zone_i)$). If $post(Zone_i)$ intersects with $post(Zone_j)$, then it is a deadlock situation (Deadlock situation 4 $\in DS$).
- If $pre(Zone_i)$ intersects with $post(Zone_j)$, then this situation is the Deadlock risk$_{post}$($R_i$, $R_j$) or the Deadlock risk$_{pre}$($R_j$, $R_i$) situations. So, in both situations $R_j$ must wait for $R_i$ according to Lemmas. [1, 2]. This situation is not a deadlock situation.
- If $(Zone_i \cap Zone_j) \subset (motion(Zone_i) \cap motion(Zone_j))$, then either $R_i$ waits for $R_j$ or $R_j$ waits for $R_i$. So, this situation is not a deadlock situation.

- If $motion(Zone_i)$ intersects with $pre(Zone_j)$, then it is the Deadlock risk$_{pre}(R_i, R_j)$ situation. According to Lemma. 1 Deadlock risk$_{pre}(R_i, R_j)$ is not a deadlock situation.
- If $motion(Zone_i)$ intersects with $pre(Zone_j)$, and $motion(Zone_j)$ intersects with $pre(Zone_i)$, then it is a deadlock situation (Deadlock situation 1$\in DS$).
- If $motion(Zone_i)$ intersects with $post(Zone_j)$, then it is the Deadlock risk$_{post}(R_i, R_j)$ situation. According to Lemma. 2 Deadlock risk$_{post}(R_i, R_j)$ is not a deadlock situation.
- If $motion(Zone_i)$ intersects with $post(Zone_j)$, and $motion(Zone_j)$ intersects with $post(Zone_i)$, then it is a deadlock situation (Deadlock situation 2 $\in DS$).
- If $post(Zone_i)$ intersects with $post(Zone_j)$, then it is a deadlock situation (Deadlock situation 4 $\in DS$).
- If $motion(Zone_i)$ intersects with both $pre(Zone_j)$ and $post(Zone_j)$, then it is a deadlock situation (Deadlock situation 3 $\in DS$).

We prove that, a situation where a requested zone intersects with the pre-motion zone of a robot that has not requested a zone, is a deadlock situation, and a situation where a robot $R_i$ must wait for a robot $R_j$, and $Zone_i$ intersects with $post(Zone_j)$, is a deadlock situation.

- If a robot $R_j$ has not requested a zone, then no wait-for relation between $R_j$ and a robot $R_i$ is established. A robot $R_i$ initially knows the position ($pre(Zone_j)$) of each robot $R_j$ in the system. So, if a requested zone $Zone_i$ intersects with the pre-motion zone of a robot $R_j$ that has not requested a zone, then $R_i$ can not own $Zone_i$, otherwise a collision between $R_i$ and $R_j$ occurs. Therefore, this situation is a deadlock situation, since neither $R_i$ nor $R_j$ can move.
- If a requested zone by a robot $R_i$ intersects with the *post-motion* zone of a robot $R_j$, then $R_j$ must wait for $R_i$ (Lemma. 2), otherwise, a collision may occur between $R_i$ and $R_j$. Therefore, if a requested zone by a robot $R_i$ intersects with the *post-motion* zone of a robot $R_j$, and $R_i$ must wait for $R_j$, then a deadlock situation occurs between $R_i$ and $R_j$.

Consequently, the possible deadlock situations are as follows. A situation that belongs to the set $DS = \{$Deadlock situation 1, Deadlock situation 2, Deadlock situation 3, Deadlock situation 4$\}$, a situation where a requested zone intersects with the pre-motion zone of a robot that has not requested a zone, and a situation where a robot $R_i$ must wait for a robot $R_j$, and $Zone_i$ intersects with $post(Zone_j)$. □

**Theorem 1 (Non triviality)** *An exception is raised only when a deadlock situation occurs.*

**Proof.** An exception is raised only by the Deadlock Handler algorithm (Algorithm. 5, line. 10). The Deadlock Handler returns an exception when it does not find a solution to resolve a deadlock situation.

The Deadlock Handler is called by the Deadlock detector algorithm (Algorithm. 4, lines. [3, 11]). In Algorithm. 4, line. 3, the situation is that, a requested zone intersects with the *pre-motion* zone of a robot that has not requested a zone. According to Lemma. 3, this situation is a deadlock situation.

In Algorithm. 4, line. 11, the situation belongs to the set $DS$, so it is a deadlock situation according to Lemma. 3.

The Deadlock Handler is also called by the Append-Graphs algorithm (Algorithm. 3, line. 9), when a robot $R_y$ must wait for a robot $R_x$, and $Zone_y$ intersects with the *post-motion* zone $post(Zone_x)$. This is a deadlock situation according to Lemma. 3.

Consequently, an exception is raised only when a deadlock situation occurs. □

**Lemma 4** *The wait-for graph $Dag_{arbiter}$ has no cycles.*

**Proof.** The algorithm Arbiter starts from a totally ordered set of elements $list$. Let us consider two different elements of $list$ $(R_i, Zone_i)$ and $(R_j, Zone_j)$. These two elements have two different indexes, assuming that the index of $(R_j, Zone_j)$ is larger than the index of $(R_i, Zone_i)$. According to the Arbiter algorithm Algorithm. 2, if $Zone_i$ intersects with $Zone_j$ then a directed edge is added to the graph $Dag_{Arbiter}$ from the element of the higher index to the element of lower index in $list$. Thus a directed edge from $(R_j, Zone_j)$ to $(R_i, Zone_i)$. Each index in $list$ corresponds to one and only one element $\in list$. If a directed edge from vertex($R_j$) to vertex($R_i$) $\in Dag_{arbiter}$ then it is impossible that a directed edge from vertex($R_i$) to vertex($R_j$) belongs to $Dag_{arbiter}$.

The previous arguments are valid for more than two elements. Let us consider the case of three elements. $(R_i, Zone_i)$, $(R_j, Zone_j)$, $(R_k, Zone_k) \in list$ and assume that each zone intersects with the two other zones, and index$(R_i, Zone_i)$ < index$(R_j, Zone_j)$ < index$(R_k, Zone_k)$. According to Algorithm. 2 the following directed edges belong to the graph $Dag_{arbiter}$. A directed edge from vertex($R_k$) to vertex($R_j$), a second directed edge from vertex($R_j$) to vertex($R_i$), and a third directed edge from vertex($R_k$) to vertex($R_i$). Since, a total order relation is *transitive* then it is impossible that a directed edge from vertex($R_i$) to vertex($R_k$) belongs to $Dag_{arbiter}$. □

**Lemma 5** *The wait-for graph $Dag_{wait}$ has no cycles.*

**Proof.** The wait-for graph $Dag_{wait}$ results from appending the graphs generated by the Arbiter algorithm, in successive batches of the protocol. According to lemma. 4 a graph $Dag_{arbiter}$ has no cycles. Appending Arbiter graphs $Dag_{arbiter}$ is managed by the Append-Graphs algorithm (Algorithm. 3) that removes the vertices which represent the robots that have released their previous zones, and then Algorithm. 3 appends the Arbiter graph $Dag_{arbiter}$ that corresponds to the requests of zones.

Consider two graphs $Dag_{arb1}$ generated in a batch number $r$ and $Dag_{arb2}$ generated in the next batch $r + 1$.

The Append algorithm, at first, updates $Dag_{arb1}$ by removing the vertices of robots that released their previous zones. When a robot $R_i$ requests a zone, $R_i$ must release the previous zone. All the robots in the system update the wait-for graph $Dag_{wait}$ consistently, by removing the same set of vertices corresponding to the robots that have released their previous zones. The consistency result is provided by the total order broadcast. (Algorithm. 15, line. 4).

$\forall$ vertex($R_i$), if vertex($R_i$) $\in Dag_{arb2} \Rightarrow$ vertex($R_i$) $\notin Dag_{arb1}$. Because, $R_i$ must release PREVIOUS($RelZone_i$) before requesting $Zone_i$, thus vertex($R_i$) is removed from $Dag_{arb1}$.

$\forall$ vertex($R_i$), if vertex($R_i$) $\in Dag_{arb1} \Rightarrow$ vertex($R_i$) $\notin Dag_{arb2}$. Because $R_i$ has not yet released its zone, thus $R_i$ can not request a new zone. Consequently, the vertices of $Dag_{arb1}$ and $Dag_{arb2}$ are distinct vertices.

The appending mechanism relies on adding a directed edge from a vertex of $Dag_{arb2}$ to a vertex of $Dag_{arb1}$, if the requested zones by the corresponding robots are intersecting zones. The appending mechanism implies the impossibility that an edge is directed from a vertex($R_k$) $\in Dag_{arb1}$ to a vertex($R_j$) $\in Dag_{arb2}$. Therefore, if the requested zone by a robot $R_i$ such that vertex($R_i$) $\in Dag_{arb2}$ intersects with $Zone_j$ of a robot $R_j$ such that vertex($R_j$) $\in Dag_{arb1}$, then $R_i$ must wait for $R_j$. Consequently, no cycles can be created by appending $Dag_{arb2}$ to the graph $Dag_{arb1}$. So, the wait-for graph $Dag_{wait}$ has no cycle. □

**Theorem 2 ( Mutual Exclusion)** *If a requested zone $Zone_i$ of $R_i$ intersects with a requested zone $Zone_j$ of $R_j$ then exclusively either $R_i$ or $R_j$ becomes the owner of its requested zone.*

$(Zone_i \cap Zone_j \neq \emptyset) \Rightarrow (R_i \text{ owns } Zone_i)$ XOR $(R_j \text{ owns } Zone_j)$

**Proof.** If a requested zone $Zone_i$ of a robot $R_i$ intersects with a zone $Zone_j$ of a robot $R_j$, then one of them waits for the other, because a directed edge is added to the graph $Dag_{arbiter}$ from the request of the higher index to the request of lower index in the list DELIVERED. (Algorithm. 2, line. 12).

Let us consider that $R_i$ waits for $R_j$, so $R_j$ releases $RelZone_j$, after that $R_i$ owns $Zone_i$. When the robot $R_i$ is the owner of $Zone_i$, the robot $R_j$ is deprived from its ownership to the zone $Zone_j$.

The robot $R_j$ just keeps a part of $post(Zone_j)$ under its reservation. $Zone_i$ does not intersect with the part of $post(Zone_j)$ that still reserved by $R_j$, because:

1. $pre(Zone_i) \cap post(Zone_j) = \emptyset$ (Proof by contradiction). If $pre(Zone_i)$ intersects with $post(Zone_j)$, then this situation is the Deadlock risk$_{pre}(R_j, R_i)$ or the Deadlock risk$_{post}(R_i, R_j)$ situations. In both situations $R_j$ must wait for $R_i$ according to Lemmas. [1, 2]. (Algorithm. 2, line. 9). This leads to a contradiction, since the assumption is that $R_i$ waits for $R_j$.

2. $motion(Zone_i) \cap post(Zone_j) = \emptyset$ (Proof by contradiction). If the *motion* zone of $R_i$ intersects with the *post-motion* zone of $R_j$, then the situation is: Deadlock risk$_{post}(R_i, R_j)$. Thus, $R_j$ must wait for $R_i$ which leads to a contradiction.

3. $post(Zone_i) \cap post(Zone_j) = \emptyset$ (Proof by contradiction). If the *post-motion* zones intersect, then the situation is a deadlock situation (Deadlock situation $4 \in DS$), which leads to a contradiction.

The graph $Dag_{arbiter}$ is generated based on the list DELIVERED , hence the *same* list (composition and order of requests) is delivered by all the robots. $Dag_{arbiter} = Arbiter(\text{DELIVERED})$. The Arbiter algorithm (Algorithm. 2) performs deterministic computations starting from the list DELIVERED, by adding a directed edge from the vertex that represents the robot of the higher request index in DELIVERED to a vertex of lower request index, if the requested zones intersect. If a deadlock situation is detected, then a request preemption policy is applied by the Deadlock Handler algorithm, such that the request that has the higher index is preempted. So, the Arbiter algorithm defines a deterministic function denoted by: *Arbiter*().

Consequently, all robots generate the same wait-for graph $Dag_{wait}$, and the ownership of intersecting zones satisfies the mutual exclusion property. □

**Theorem 3 (Liveness)** *If a robot $R_i$ requests $Zone_i$ then eventually ($R_i$ owns $Zone_i$ or an exception is returned).*

$R_i$ requests $Zone_i \Rightarrow \Diamond$ ($R_i$ owns $Zone_i$ or Exception)

**Proof.** If a robot $R_i$ requests a zone $Zone_i$, then:

1. If $Zone_i$ does not intersect with a zone $Zone_j$, then $R_i$ owns $Zone_i$.

2. If $Zone_i$ intersects with a zone $Zone_j$, then a directed edge is created between vertex($R_i$) and vertex($R_j$) in the wait-for graph $Dag_{wait}$. According to Lemma. 5 the graph $Dag_{wait}$ has no cycles. Therefore, $R_i$ eventually owns $Zone_i$.

3. If a deadlock situation is detected, then the Deadlock Handler is called. If the Deadlock Handler algorithm does not find a solution to resolve a deadlock situation, then an exception is raised by the Deadlock Handler (Algorithm. 5, line. 10).

Therefore, $R_i$ requests $Zone_i \Rightarrow \Diamond$ ($R_i$ owns $Zone_i$ or Exception). □

## 5.3 Fault-tolerant collision prevention protocols

We consider the crash of robots. If a robot $R_j$ has crashed then, there exist some robots that are blocked waiting for $R_j$ to release $Zone_j$. The set of robots that are blocked waiting for $R_j$ gets larger as the time progresses (*Snowball* effect).

Therefore, it is necessary to provide collision prevention protocols that tolerates the crash of some robots of the system, and allows the system of robots to progress toward their goals.

In this Section, we provide two fault-tolerant collision prevention protocols, the first is preemptive in the sense that a request of a robot $R_j$ is preempted if $R_j$ is considered as a crashed robot. $R_j$ is considered crashed if it is suspected by the majority of robots in the system. The second protocol is non-preemptive, so a request of a robot $R_j$ is not preempted, instead a robot $R_i$ cancels a request ($R_i$, $Zone_i$) if $R_i$ suspects $R_j$ and $R_i$ waits for $R_j$ ($Zone_i$ intersects with $Zone_j$).

### 5.3.1 Failure model

We consider that a robot can fail by crash and that crash is permanent. A *correct* robot is defined as a robot that never crashes. A *faulty* robot is defined as a robot that might crashes.

A robot $R_i$ is provided with a $\Diamond\mathcal{S}$ failure detector $FD_i$ that triggers a SUSPICION if it *suspects* a robot $R_j$ after trusting it, $FD_i$ triggers also a TRUST if it *trusts* $R_j$ after suspecting it.

We assume that the *majority* of robots in the system are correct robots. The number of faulty robots $f$ is less than half of the robots of the system. ($f < \lceil \frac{n}{2} \rceil$), where $n$ is the total number of robots.

### 5.3.2 A preemptive fault-tolerant collision prevention protocol

The idea of the preemptive protocol is as follows. A request of a robot $R_j$ is preempted if it is considered as a crashed robot by the majority of robots, and if $R_j$ has not owned its requested zone $Zone_j$. When the request ($R_j$, $Zone_j$) is preempted, $R_j$ restarts a new request of $Zone_j$.

If $R_j$ has owned $Zone_j$ then, $Zone_j$ is considered as a blocked zone, hence a request ($R_i$, $Zone_i$) of a robot $R_i$ that waits for $R_j$ and $Zone_i$ intersects with $Zone_j$, is preempted. When the request ($R_i$, $Zone_i$) is preempted, $R_i$ requests an alternative zone.

**The preemptive protocol description**

The collision prevention protocol in the face of failures, relies on a total order broadcast algorithm, since it is fault-tolerant. The total order broadcast algorithm is defined in terms of two primitives *TO-Broadcast* and *TO-Deliver*. When requests are *TO-Delivered*, the Arbiter graph $Dag_{arbiter}$ is generated as explained in Section. 6.2.

**Status of robots** If $R_i$ *suspects* that a robot $R_j$ has crashed, (when $R_i$'s failure detector module $FD_i$ triggers a suspicion after trusting $R_j$) then, $R_i$ *TO-Broadcasts* a message indicating that $R_j$ is suspected as a crashed robot.[1]

When $R_i$'s failure detector module $FD_i$ triggers a trust after suspecting $R_j$) then, $R_i$ *TO-Broadcasts* a message indicating that $R_j$ is trusted again.

When a robot $R_i$ *TO-Delivers* $\lceil \frac{n+1}{2} \rceil$ messages from different senders (majority of robots), indicating that a robot $R_j$ is suspected as crashed, $R_i$ sets the *status* of $R_j$ to "crashed" (initially, the status of a robot is set to the value "not-crashed").

If a robot $R_i$ *TO-Delivers* a *trust* message concerning a robot $R_j$ then, the status of $R_j$ is reset to the value "not-crashed".

**Behavior of a robot** $R_i$ If $R_i$ *TO-Delivers* $\lceil \frac{n+1}{2} \rceil$ messages from different senders (majority of robots), indicating that $R_i$ is suspected crashed, and $R_i$ has not owned $Zone_i$, then the request of $R_i$ is preempted and $R_i$ restarts its request of $Zone_i$.

If $R_i$ *TO-Delivers* $\lceil \frac{n+1}{2} \rceil$ messages from different senders (majority of robots) indicating that $R_i$ is suspected crashed, and $R_i$ has owned $Zone_i$, then $R_i$ ignores the messages, since other robots consider $Zone_i$ as a "blocked" zone and owned always by $R_i$. When $R_i$ reaches the *post-motion* zone $post(Zone_i)$, $R_i$ releases $Zone_i$. When a robot $R_j$ *TO-Delivers* the release message of $R_i$, $R_j$ resets the status of $Zone_i$ to the value "unblocked" and resets the status of $R_i$ to the value "not-crashed".

If the robot $R_i$ has actually crashed, then $Zone_i$ is held blocked. Because, it is impossible for a robot $R_j$ to decide *deterministically* whether $R_i$ has crashed or not, in an *asynchronous* distributed system. [14].

When a robot $R_i$ receives a release message from all the robots $R_j$ that $R_i$ waits for, it invokes the procedure Reserve($Zone_i$).

**Reserve($Zone_i$)** A robot $R_i$ starts the procedure Reserve($Zone_i$), when all the robots $R_j$ that $R_i$ waits for, release their zones $Zone_j$. In this preemptive protocol, a robot reserves a zone via Total Order Broadcast, to ensure that the robots decide consistently whether a robot $R_i$

[1]The suspicion of $R_j$ does not mean that $R_j$ has crashed.

has owned $Zone_i$ when $R_i$ is suspected by the majority of robots in the system. So, $R_i$ *TO-Broadcasts* a message indicating that $R_i$ reserves $Zone_i$. When a robot $R_i$ *TO-Delivers* a message RESERVE($Zone_i$), $R_i$ owns $Zone_i$ and sets the status of $R_i$ to "not-crashed". All robots agree that $R_i$ owns $Zone_i$. Therefore, if the status of $R_i$ changes to "crashed" after $R_i$ has owned $Zone_i$ then, $Zone_i$ is considered as a "blocked" zone and owned always by $R_i$.

**Crash Handler** The Crash Handler of a robot $R_i$ is called, when the status of a robot $R_j$ becomes "crashed", and $R_j$ has owned $Zone_j$. In such a situation, $Zone_j$ is handled as a "blocked" zone owned always by $R_j$, since $R_i$ cannot decide deterministically whether $R_j$ has really crashed or not. If $R_j$ has not really crashed then, the status of $Zone_j$ is reset to "unblocked" when $R_j$ eventually releases $Zone_j$.

- If $Zone_i$ intersects with $Zone_j$ (i.e., $R_i$ waits for $R_j$ *directly*) then, the request ($R_i$, $Zone_i$) is preempted and the graph $Dag_{wait}$ is updated by removing the vertex($R_i$) and its related (incoming and outgoing) edges from $Dag_{wait}$. Also, for all $R_k$ that conflict with $R_j$ ($Zone_k$ intersects with$Zone_j$), the vertex($R_k$) with its related edges is removed from $Dag_{wait}$.

  The Crash Handler of $R_i$ offers an alternative zone $Zone_i^{alternative}$ to the robot $R_i$, which starts a request of the alternative zone. REQUEST($Zone_i^{alternative}$). If there is no available alternative chunk, then the Crash Handler algorithm raises an Exception.

- If $Zone_i$ does not intersect with $Zone_j$ then, the wait-for graph $Dag_{wait}$ is updated by removing vertex($R_k$) with its related (incoming and outgoing) edges, for all robots $R_k$ such that $Zone_k$ intersects with $Zone_j$. Then $R_i$ continues the execution of the protocol.

**Deadlock detector** The deadlock detector, in presence of failures, is additionally, required to check the intersection of a requested zone with a "blocked" zone. The status of a zone $Zone_j$ is set to the value "blocked" when the status of the owner $R_j$ becomes "crashed" and $R_j$ has owned $Zone_j$. The status of a zone is remained "blocked" until a RELEASE($Zone_j$) message of the robot $R_j$ is *TO-delivered*.

**Append-Graphs** The Append-Graphs algorithm, in presence of failures, is required to update the status of a "blocked" zone $Zone_j$. The Append-Graphs algorithm sets the status of $Zone_j$ to the value "unblocked". The algorithm Append-Graphs is responsible for handling a released zone, by removing the corresponding vertex and its related edges from the graph $Dag_{wait}$ before appending the graph $Dag_{arbiter}$. Thus, Append-Graphs algorithm unblocks a zone, when the owner releases the blocked zone.

**Algorithm 6** Fault tolerant Collision prevention protocol (Code for robot $R_i$)

```
1:  Initialisation:
2:     PREVIOUS(RelZone_i) := ⊥; Dag_wait := ⊥; DELIVERED := ∅;
3:     for all robots R_j do
4:        Status(R_j) := "not-crashed"
5:        count[R_j] := 0
6:     end for

7:  procedure Request(Zone_i)
8:     TO-broadcast [REQUEST, Zone_i, RELEASE, PREVIOUS(RelZone_i)]

9:     when TO-Deliver [REQUEST, Zone_j, RELEASE, PREVIOUS(RelZone_j)]
10:       DELIVERED := DELIVERED ∪ (REQUEST, Zone_j, RELEASE, PREVIOUS(RelZone_j))
11:       Dag_arbiter := Arbiter(DELIVERED)
12:       Dag_wait := Append-Graphs(Dag_wait, Dag_arbiter, DELIVERED)

13:       when the vertex of R_i in Dag_wait becomes a sink vertex (has no outgoing edges)
14:          return                    {all robots R_j that R_i waits for, has released their zones}
15:       end when
16:    end when
17: end Request(Zone_i)

18: procedure Reserve(Zone_i)
19:    TO-Broadcst(RESERVE(Zone_i))
20: end Reserve(Zone_i)

21: task Status(R_i)
22:    when TO-Deliver(m)
23:       if m is SUSPECT(R_i) then
24:          TO-Broadcast(TRUST, R_i)        {R_i TO-broadcasts a message to indicate that R_i has not crashed}
25:          count[R_i] := count[R_i] + 1
26:          if count[R_i] > ⌈(n+1)/2⌉ and R_i does not own Zone_i then
27:             update the wait-for graph Dag_wait by removing vertex(R_i) and its related edges
28:             Restart(R_i, Zone_i)                {R_i restarts the procedure Request(Zone_i)}
29:             count[R_i] := 0
30:             Status(R_i) := "not-crashed"
31:          end if
32:       end if
33:       if m is TRUST(R_i) then
34:          count[R_i] := 0                {resets the counter when R_i is trusted}
35:          Status(R_i) := "not-crashed"
36:       end if
37:       if m is RESERVE(Zone_i) then
38:          R_i owns Zone_i
39:          count[R_i] := 0                {all robots agree that R_i owns Zone_i}
40:          Status(R_i) := "not-crashed"
41:          when R_i reaches the post-motion zone post(Zone_i)
42:             PREVIOUS(RelZone_i) := Zone_i except the place that R_i may possibly occupy.
43:          end when
44:       end if
45:    end when
46: end
```

**Algorithm 7** Status algorithm

```
1:  task Status(R_j)
2:    if (Status(R_j) ="not-crashed") and suspect(R_j) then
3:      TO-Broadcast(SUSPECT, R_j) {R_i TO-broadcasts a message that R_j is suspected, when FD_i suspects R_j}
4:    end if
5:    if (Status(R_j) ="crashed") and trust(R_j) then
6:      TO-Broadcast(TRUST, R_j)        {R_i TO-broadcasts a message that R_j is trusted, when FD_i trusts R_j}
7:    end if

8:    when TO-Deliver(m)
9:      if m is SUSPECT(R_j) then
10:       count[R_j] := count[R_j] + 1
11:       if count[R_j] > ⌈(n+1)/2⌉ then
12:         Status(R_j) := "crashed"
13:       end if
14:     end if
15:     if m is TRUST(R_j) then
16:       count[R_j] := 0                                   {resets the counter when R_j is trusted}
17:       Status(R_j) := "not-crashed"
18:     end if
19:     if m is RESERVE(Zone_j) then
20:       Status(R_j) := "not-crashed"
21:       count[R_j] := 0                {sender(m) is R_j, thus the Status of R_j is updated by all robots}
22:       R_j owns Zone_j                                   {All robots agree that R_j owns Zone_j}
23:     end if
24:     if (Status(R_j) ="crashed") and (R_j does not own Zone_j) then
25:       Request Preemption of (R_j, Zone_j)               {The request of (R_j, Zone_j) is preempted}
26:       update the wait-for graph Dag_wait by removing vertex(R_j) and its related edges
27:       count[R_j] := 0
28:       Status(R_j) := "not-crashed"
29:     end if
30:     if (Status(R_j) = "crashed") and (R_j owns Zone_j) then
31:       Crash-Handler(R_j, Zone_j)                        {In such a situation, Zone_j is held owned by R_j}
32:     end if
33:   end when
34: end
```

**Algorithm 8** Restart algorithm

```
1: procedure Restart(R_i, Zone_i)
2:   Request Preemption (R_i, Zone_i)   {The request (R_i, Zone_i) is preempted, thus R_i must restart the request of
     Zone_i}
3:   update the wait-for graph Dag_wait by removing vertex(R_i) and its related edges
4:   Request(Zone_i)                   {R_i restarts the protocol by Request(Zone_i) with PREVIOUS(RelZone_i)=⊥}
5: end
```

**Algorithm 9** Deadlock detector algorithm

```
1:  function Deadlock-detector (list)
2:     if Zone_i ∈ list intersects with "blocked" zone then
3:        return true
4:     end if

5:     if Zone_i ∈ list intersects with pre(Zone_j) of R_j that does not have a request in list then
6:        Deadlock Handler(R_x, Zone_x, R_y, Zone_y)
7:     end if

8:     for all (R_x, R_y) ∈ list do
9:        Deadlock situation 1:=
          [Zone_x ∩ pre(Zone_y) ≠ ∅] and [Zone_y ∩ pre(Zone_x) ≠ ∅] and [post(Zone_x) ∩ post(Zone_y) = ∅]
10:       Deadlock situation 2:=
          [Zone_x ∩ post(Zone_y) ≠ ∅] and [Zone_y ∩ post(Zone_x) ≠ ∅] and [post(Zone_x) ∩ post(Zone_y) = ∅]
11:       Deadlock situation 3:=
          [Zone_x ∩ pre(Zone_y) ≠ ∅] and [Zone_x ∩ post(Zone_y) ≠ ∅] and [post(Zone_x) ∩ post(Zone_y) = ∅]
12:       Deadlock situation 4:=
          post(Zone_x) ∩ post(Zone_y) ≠ ∅

13:       if Deadlock situation 1 or Deadlock situation 2 or Deadlock situation 3 or Deadlock situation 4 then
14:          Deadlock Handler(R_x, Zone_x, R_y, Zone_y)
15:       end if
16:    end for
17: end
```

**Algorithm 10** Crash Handler algorithm

```
1:  function Crash-Handler(R_j, Zone_j)
2:     status(Zone_j) := "blocked"          {all robots update the status of Zone_j to "blocked"}
3:     for all R_k such that Zone_k intersects with Zone_j do
4:        update the wait-for graph Dag_wait by removing vertex(R_k) and its related edges.
5:     end for

6:     if Zone_i intersects with Zone_j then
7:        Request Preemption (R_i, Zone_i)          {The request (R_i, Zone_i) is canceled}
8:        if no possible alternative chunk then
9:           throw Exception                        {There is no solution}
10:          return Exception
11:       end if

12:       Zone_i := Zone_i^alternative              {The Crash Handler proposes an alternative chunk for R_i}
13:       Request (Zone_i)
14:    end if
15: end
```

**Algorithm 11** Append-Graphs

```
1: function Append-Graphs(Dag_wait, Dag_arbiter, list)
2:   for all (RELEASE, PREVIUOUS(RelZone_j) ≠⊥) ∈ list do
3:     if status(PREVIUOUS(RelZone_j)) = "blocked" then
4:       status(PREVIUOUS(RelZone_j)) := "unblocked"
5:       remove the vertex(R_j) from Dag_wait
6:       count[R_j] := 0                                   {R_j has not crashed}
7:       Status(R_j) := "not-crashed"
8:     else
9:       remove the vertex(R_j) and its incoming edges from Dag_wait
10:    end if
11:  end for

12:  for all R_y ∈ Dag_arbiter do
13:    for all R_x ∈ Dag_wait do
14:      if Zone_y intersects with Zone_x then
15:        if Zone_y intersects with post(Zone_x) then
16:          Deadlock-Handler(R_x, Zone_x, R_y, Zone_y)   {R_y must wait-for R_x, but if Zone_y intersects with
             post(Zone_x) then R_y calls the Deadlock-Handler}
17:        else
18:          Dag_wait := Dag_wait ∪ DirEdge(R_y, R_x)     {R_y must wait-for R_x}
19:        end if
20:      end if
21:    end for
22:  end for
23:  return Dag_wait
24: end
```

### Proof of correctness for the preemptive fault-tolerant protocol

We prove that the preemptive protocol satisfies the *safety* property (Mutual Exclusion), and the Non-Triviality property, also we prove that the Liveness property holds if robots are provided with a failure detector of class $\diamond\mathcal{P}$.

**Theorem 4 ( Mutual Exclusion)** *If a requested zone $Zone_i$ of $R_i$ intersects with a requested zone $Zone_j$ of $R_j$ then exclusively either $R_i$ or $R_j$ becomes the owner of its requested zone.*

$(Zone_i \cap Zone_j \neq \emptyset) \Rightarrow (R_i \text{ owns } Zone_i)$ XOR $(R_j \text{ owns } Zone_j)$

**Proof.** The total order broadcast algorithm, guarantees that all robots *TO-Deliver* the requests and the reservations in the same total order, in presence of failures (crash of robots).

The protocol relies on a $\diamond\mathcal{S}$ failure detector, with majority of correct robots. The number of faulty robots $f$ is less than half of the robots of the system. ($f < \lceil \frac{n}{2} \rceil$), where $n$ is the total number of robots. The problem of total order broadcast is *equivalent* to the consensus problem, and it has been shown in [6], that $\diamond\mathcal{S}$ is the weakest class of failure detector that allows solving the consensus problem in an asynchronous system prone to process crashes, if the majority of processes are correct.

If a robot $R_i$ suspects a robot $R_j$, then $R_i$ *TO-Broadcasts* a suspicion message that $R_j$ has crashed. All robots *TO-Delivers* suspicion messages that $R_j$ has crashed in the same total order.

When $R_i$ *TO-Delivers* ($\lceil \frac{n+1}{2} \rceil$) suspicion messages that $R_j$ has crashed, there are two situations:

- If $R_j$ has not owned $Zone_j$, then the request of $R_j$ is preempted.
  - If $R_j$ has not really crashed, then it restarts a request of $Zone_j$.
  - If $R_j$ has really crashed, then it just occupies the $pre(Zone_j)$. If a robot $R_i$ requests a zone $Zone_i$ that intersects with $pre(Zone_j)$, then the Deadlock detector detects this situation (Algorithm. 9, line. 6), hence the Deadlock Handler is called to resolve this deadlock situation by proposing an alternative chunk to $R_i$.
- If $R_j$ has owned $Zone_j$, then $Zone_j$ is considered as a blocked zone. Hence, no robot other than $R_j$ can own $Zone_j$.
  - If $R_j$ has not really crashed, then $R_j$ releases the blocked zone when $R_j$ reaches $post(Zone_j)$.
  - If $R_j$ has really crashed, then $Zone_j$ remains blocked.

The ownership of $Zone_j$ by $R_j$ is decided consistently by all robots (procedure Reserve($Zone_j$)), since the reserving a zone is performed relying on the total order broadcast.

Consequently, the ownership of intersecting zones satisfies the mutual exclusion property, and the *safety* property is satisfied. □

**Theorem 5 (Liveness)** *If a robot $R_i$ is a correct robot, $R_i$ requests $Zone_i$ then eventually ($R_i$ owns $Zone_i$ or an exception is returned), with the additional assumption that robots are provided with $\diamond\mathcal{P}$ failure detector. The* Liveness *property does not hold if robots are provided with $\diamond\mathcal{S}$ failure detector.*

$R_i$ requests $Zone_i \Rightarrow \diamond$ ($R_i$ owns $Zone_i$ or Exception), with the additional assumption that robots are provided with $\diamond\mathcal{P}$ failure detector.

**Proof.** Robots are provided with $\diamond\mathcal{P}$ failure detector. The STRONG COMPLETENESS property of the failure detector, implies that there is a time after which every faulty robot is permanently suspected by all correct robots. Therefore, eventually $R_i$ (correct robot) suspects permanently a faulty robot $R_j$.

If $R_i$ is suspected by the majority of robots ($> \lceil \frac{n+1}{2} \rceil$), then the request of $R_i$ is preempted, and $R_i$ restarts its request of $Zone_i$. Robots are provided with a $\diamond\mathcal{P}$ failure detector (STRONG

COMPLETENESS and EVENTUAL STRONG ACCURACY). The EVENTUAL STRONG ACCURACY property of the failure detector, implies that, there is a time after which correct robots are not suspected by any correct robot. Thus, if $R_i$ is a correct robot, then eventually, it will not be suspected by any correct robot. Since, there are at least $\lceil \frac{n+1}{2} \rceil$ correct robots in the system, then eventually $R_i$ will not be suspected by the majority of robots, so eventually, $R_i$ owns a requested zone $Zone_i$.

If $R_i$ waits for a robot $R_j$, such that $R_j$ is suspected, as a crashed robot, by the majority of robots, then:

- If $R_j$ has not owned $Zone_j$, then the request of $R_j$ is preempted, and consequently, the wait-for relation between $R_i$ and $R_j$ is canceled.
- If $R_j$ has owned $Zone_j$, then $Zone_j$ is considered as a blocked zone. There are two situations:
  - If $Zone_i$ intersects with $Zone_j$ then, the Crash Handler preempts the request ($R_i$, $Zone_i$) (Algorithm. 10, line. 7), and then $R_i$ restarts a request of an alternative zone. If the Crash Handler does not find an alternative chunk, then an exception is returned. (Algorithm. 10, line. 9).
  - If $Zone_i$ does not intersect with $Zone_j$ then, the wait-for graph $Dag_{wait}$ is updated, and a vertex($R_k$) is removed with its edges, for robots $R_k$ such that $Zone_k$ intersects with $Zone_j$. Then, $R_i$ continues the execution of the protocol.

Therefore, if a robot $R_i$ is a correct robot, $R_i$ requests $Zone_i \Rightarrow \Diamond$ ($R_i$ owns $Zone_i$ or Exception), with the additional assumption that robots are provided with $\Diamond\mathcal{P}$ failure detector.

The *Liveness* property does not hold if robots are provided with $\Diamond\mathcal{S}$ failure detector, because the EVENTUAL WEAK ACCURACY property implies that, there is a time after which *some* correct robot is never suspected by any correct robot. So, the ACCURACY property does not hold for all correct robots, and a correct robot might be suspected by correct robots. □

**Lemma 6** *The possible deadlock situations are the same deadlock situations presented in Lemma. 3, (failure-free protocol) with the additional deadlock situation, that occurs if a requested zone intersects with a "blocked" zone.*

**Proof.** If a requested zone $Zone_i$ intersects with a blocked zone $Zone_j$, then this situation is a deadlock situation. Because, the robot $R_j$ is *suspected* as a crashed robot, and it is, deterministically impossible to distinguish a crashed robot from a very slow one, in asynchronous distributed systems prone to crash failures. Thus, $Zone_j$ is considered as a "blocked" zone for $R_j$, and $Zone_i$ that intersects with $Zone_j$ creates a deadlock situation. □

**Theorem 6 (Non triviality)** *An exception is raised only when a deadlock situation occurs.*

**Proof.** An exception is raised by the Deadlock Handler algorithm (the same as for the failure-free protocol). The Deadlock Handler returns an exception when it does not find a solution to resolve a detected deadlock situation.

Additionally, the fault-tolerant protocol raises an exception when the Crash Handler module does not find a solution for the deadlock situation presented in Lemma. 6, where a requested zone intersects with a "blocked" zone. (Algorithm. 10, line. 9).

Consequently, an exception is raised only when a deadlock situation occurs. □

### 5.3.3 A non-preemptive fault-tolerant collision prevention protocol

The non-preemptive fault-tolerant collision prevention protocol is based on the following idea. If a robot $R_i$ waits *directly* for a robot $R_j$ ($Zone_i$ intersects with $Zone_j$), and $R_i$ suspects that $R_j$ has crashed, then $R_i$ cancels the request ($R_i$, $Zone_i$) and requests an alternative zone that does not intersect with $Zone_j$. The difference with the previous fault-tolerant protocol is that, the request of a robot $R_i$ can not be preempted due to the vote of other robots in the system, however $R_i$ cancels its request of $Zone_i$ (due to a decision of $R_i$) when $R_i$ suspects the robot $R_j$.

#### Proof of correctness for the non-preemptive protocol

We prove that the non-preemptive fault-tolerant protocol satisfies the properties, *safety* property (Mutual Exclusion), the Liveness property, and the Non-Triviality property.

**Theorem 7 ( Mutual Exclusion)** *If a requested zone $Zone_i$ of $R_i$ intersects with a requested zone $Zone_j$ of $R_j$ then exclusively either $R_i$ or $R_j$ becomes the owner of its requested zone.*

$(Zone_i \cap Zone_j \neq \emptyset) \Rightarrow (R_i \text{ owns } Zone_i)$ XOR $(R_j \text{ owns } Zone_j)$

**Proof.** If a robot $R_i$ suspects a robot $R_j$ and $Zone_i$ intersects with $Zone_j$ then, $R_i$ cancels the request ($R_i$, $Zone_i$) and then the Alternative Handler algorithm proposes an alternative chunk for $R_i$ that does not intersect with $Zone_j$.

Therefore, the non-preemptive collision prevention protocol satisfies the safety property of the system, that no collision can occur between robots.

$(Zone_i \cap Zone_j \neq \emptyset) \Rightarrow (R_i \text{ owns } Zone_i)$ XOR $(R_j \text{ owns } Zone_j)$. □

**Theorem 8 (Liveness)** *$R_i$ is a correct robot, $R_i$ requests $Zone_i$ then eventually ($R_i$ owns $Zone_i$ or an exception is returned).*

$R_i$ requests $Zone_i \Rightarrow \Diamond$ ($R_i$ owns $Zone_i$ or Exception).

**Algorithm 12** Non-preemptive fault tolerant collision prevention protocol (Code for robot $R_i$)

```
1:  Initialisation:
2:     PREVIOUS(RelZone_i) := ⊥; Dag_wait := ⊥; DELIVERED := ∅;

3:  procedure Request(Zone_i)
4:     TO-broadcast [REQUEST, Zone_i, RELEASE, PREVIOUS(RelZone_i)]

5:     when TO-Deliver [REQUEST, Zone_j, RELEASE, PREVIOUS(RelZone_j)]
6:        DELIVERED := DELIVERED ∪ (REQUEST, Zone_j, RELEASE, PREVIOUS(RelZone_j))
7:        Dag_arbiter := Arbiter(DELIVERED)
8:        Dag_wait := Append-Graphs(Dag_wait, Dag_arbiter, DELIVERED)

9:        if SUSPECT(R_j) and Zone_i intersects with Zone_j then
10:          Cancel request (R_i, Zone_i)
11:       end if

12:       Alternative(R_i, Zone_i)

13:       when the vertex representing R_i in Dag_wait becomes a sink vertex
14:          return
15:       end when
16:    end when
17: end Request(Zone_i)

18: procedure Reserve(Zone_i)
19:    R_i reserves Zone_i                                   {R_i owns Zone_i}
20:    when R_i reaches the post-motion zone post(Zone_i)
21:       PREVIOUS(RelZone_i) := Zone_i except the place that R_i may possibly occupy.
22:    end when
23: end Reserve(Zone_i)
```

**Algorithm 13** Cancel-Request algorithm

```
1:  procedure Cancel Request(R_i, Zone_i)
2:     TO-Broadcast(CANCEL, R_i, Zone_i)        {The request (R_i, Zone_i) is canceled}

3:     when TO-Deliver(CANCEL, R_i, Zone_i)
4:        update the wait-for graph Dag_wait by removing vertex(R_i) and its related edges
5:     end when
6:  end
```

**Algorithm 14** Alternative Handler algorithm

```
1:  function Alternative(R_i, Zone_i)
2:     if no possible alternative chunk then
3:        throw Exception                                  {There is no solution}
4:        return Exception
5:     end if

6:     Zone_i := Zone_i^alternative     {The Alternative Handler proposes an alternative chunk for R_i}
7:     Request (Zone_i)
8:  end
```

**Proof.** If a robot $R_i$ requests $Zone_i$ and $R_i$ waits for some robot $R_j$ then:

- If the failure detector $FD_i$ of $R_i$ suspects that $R_j$ has crashed and $Zone_i$ intersects with $Zone_j$ then, $R_i$ cancels its request ($R_i$, $Zone_i$) and requests an alternative zone. If there is no available alternative chunk, then an exception is raised by the Alternative Handler algorithm. (Algorithm. 14, line. 3).
- If the failure detector $FD_i$ trusts $R_j$ then, $R_i$ waits for $R_j$ until $R_j$ releases $Zone_j$.

A request ($R_i$, $Zone_i$) cannot be preempted neither by a robot $R_k$ nor by a vote of other robots in the system. The robot $R_i$ cancels the request ($R_i$, $Zone_i$).

Robots are provided with a $\diamond\mathcal{S}$ failure detector (STRONG COMPLETENESS and EVENTUAL STRONG ACCURACY). The STRONG COMPLETENESS property of the failure detector, implies that there is a time after which every faulty robot is permanently suspected by all correct robots. Therefore, eventually $R_i$ (correct robot) suspects permanently a faulty robot $R_j$ and cancels the request ($R_i$, $Zone_i$).

Consequently, if a correct robot $R_i$ requests $Zone_i$ then eventually ($R_i$ owns $Zone_i$ or an exception is returned). □

**Lemma 7** *The possible deadlock situations are the same deadlock situations presented in Lemma. 3, (failure-free model) with the additional deadlock situation, that occurs if a robot $R_i$ suspects a robot $R_j$ and $Zone_i$ intersects with $Zone_j$.*

**Proof.** If a robot $R_i$ suspects that a robot $R_j$ has crashed and $Zone_i$ intersects with $Zone_j$ then, a possible deadlock situation may occurs, if the robot $R_i$ keeps waiting for $R_j$. Because, $R_i$ cannot determine whether $R_j$ has crashed or not. □

**Theorem 9 (Non triviality)** *An exception is raised only when a deadlock situation occurs.*

**Proof.** In addition to exceptions raised by the Deadlock Handler algorithm which are discussed in the failure-free protocol (Section. 6.2, Algorithm. 5), an exception is raised by the non-preemptive fault-tolerant protocol (the Alternative Handler algorithm. Algorithm. 14, line. 3). The Alternative Handler algorithm raises an exception when it does not find an alternative chunk for a robot $R_i$ that suspects $R_j$ and $Zone_i$ intersects with $Zone_j$.

Consequently, an exception is raised only when a deadlock situation occurs. □

## 5.4 Performance analysis

We study the performance of our protocol in terms of the time needed by a robot $R_i$ to reach a given destination when robots are active (robots do not sleep), in the failure-free model. We compute the average effective speed of robots executing our collision prevention protocol. For simplicity, we assume in this section that the physical dimensions of robots are too small such that a robot can be considered as a point in the plane. The geometrical incertitude related to the positioning system, translational and rotational movement are neglected.

### 5.4.1 Time needed to reserve and move along a chunk

The average physical speed of a robot is denoted by: $V_{mot}$. We calculate the average time required for a robot $R_i$ to reserve and move along a chunk of length $D_{ch}$ with a physical speed $V_{mot}$.

When a robot requests a zone, it releases the previously owned zone thus, a robot waits at most for $(n-1)$ robots where $n$ is the number of robots of the system. So, the average number of robots that $R_i$ waits on is:

$$n_{avg} = \frac{n-1}{2} \tag{5.1}$$

**Communication delays.** The average communication delays in the system is denoted: $T_{com}$. When all the robots are active running the protocol (robots do not sleep), then the time needed to reserve and move along a chunk denoted $T_{ch}$ is computed as the sum of the time needed by each of the following steps:

1. The delay of the Total Order Broadcast algorithm denoted by: $T_{AB}$. We assume that the delay of the total order broadcast algorithm is: $T\ n$
2. The time needed for local computations by robots (to build the *wait-for* graph) is neglected.
3. The time to receive the release messages from $n_{avg}$ robots each of which has owned its zone for $\frac{D_{ch}}{V_{mot}}$ time units is: $n_{avg}(T_{com} + \frac{D_{ch}}{V_{mot}})$.
4. The time needed by $R_i$ to move along a chunk is: $\frac{D_{ch}}{V_{mot}}$.

Therefore, the time needed to reserve and move along a chunk $T_{ch}$ is:

$$T_{ch} = Tn + n_{avg}(T_{com} + \frac{D_{ch}}{V_{mot}}) + \frac{D_{ch}}{V_{mot}} \tag{5.2}$$

So,

$$T_{ch} = Tn + \frac{n-1}{2}T_{com} + (\frac{n+1}{2})\frac{D_{ch}}{V_{mot}} \tag{5.3}$$

### 5.4.2 Average effective speed

In this subsection, we compute the average effective speed $V$ of a robot $R_i$ as a function of the chunk length $D_{ch}$ and of the number of robots $n$ in the system. A robot $R_i$ makes on average $\frac{D_{trip}}{D_{ch}}$ steps to move along a path of length $D_{trip}$. The time to progress a distance $D_{trip}$ is:

$$T_{trip} = \frac{D_{trip}}{D_{ch}}[Tn + \frac{n-1}{2}T_{com} + (\frac{n+1}{2})\frac{D_{ch}}{V_{mot}}] \tag{5.4}$$

The speed $V$ is $\frac{D_{trip}}{T_{trip}}$. Thus, the average effective speed $V$ is:

$$V = \frac{D_{ch}}{Tn + \frac{n-1}{2}T_{com} + (\frac{n+1}{2V_{mot}})D_{ch}} \tag{5.5}$$

The previous relation shows that the effective speed is a function of the chunk length and the number of robots $n$ , also the effective speed depends on some system-based fixed parameters such as the communication delays $T_{com}$ and the physical speed of robots $V_{mot}$. The effective speed depends also on the performance of the Total Order Broadcast algorithm.

### 5.4.3 Average effective speed vs Chunk length

In this Subsection, we focus on the relation between the average effective speed and the chunk length for a given number of robots $n$.

The first derivative of the function effective speed with respect to the chunk length is:

$$\frac{dV}{dD_{ch}} = \frac{Tn + \frac{n-1}{2}T_{com}}{[Tn + \frac{n-1}{2}T_{com} + (\frac{n+1}{2V_{mot}})D_{ch}]^2} \tag{5.6}$$

The derivative of the effective speed with respect to the chunk length is always positive. So, the effective speed increases as the chunk length increases.

The explanation is that a robot $R_i$ waits at most for $n-1$ robots (in a static group of $n$ robots) to move along each chunk of its path. $R_i$ needs to do a certain number of steps to reach a destination, and the number of steps is a function of the chunk length. When the chunk length increases, the number of steps decreases. Therefore, the average effective speed $V$ increases with the chunk length $D_{ch}$.

Equation. 5.5 implies that the average effective speed approaches toward the value $\frac{2V_{mot}}{n+1}$ as the chunk length tends to infinity.

$$\lim_{D_{ch}\to\infty} V = \frac{2}{n+1}V_{mot} \tag{5.7}$$

Figure. 5.4 represents the relationship between the speed and the chunk length for different values of number of robots. The average effective speed of robots increases as the chunk length increases for a given number of robots, and there is an optimal value of the chunk length that

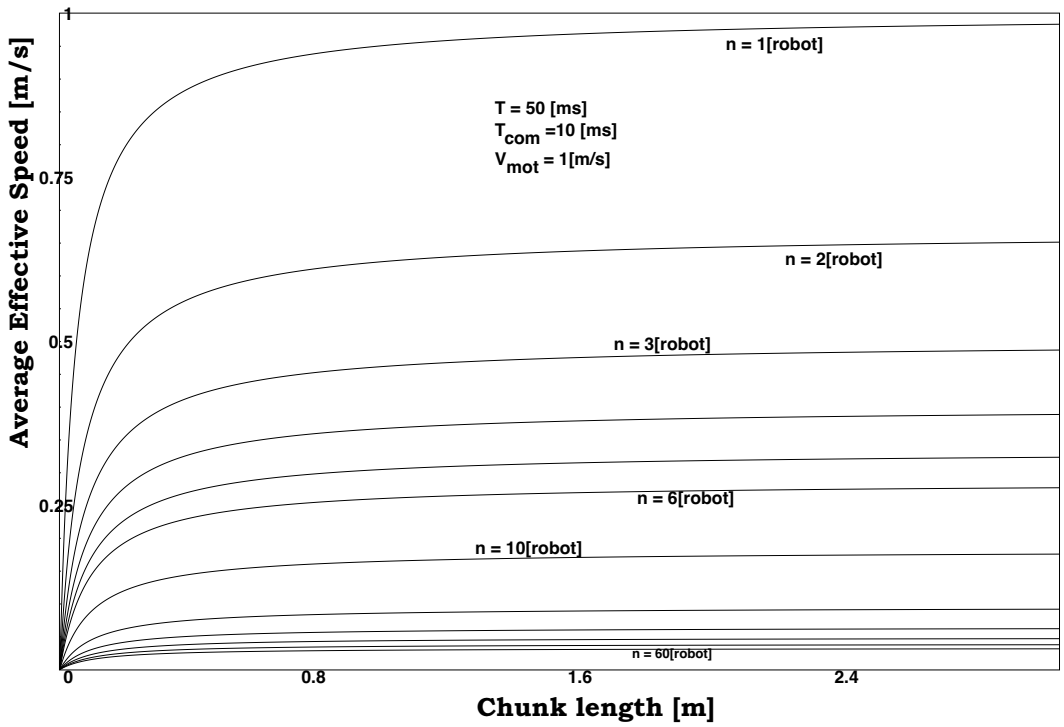



Figure 5.4: Average effective speed vs chunk length.

maximize the average effective speed for a given number of robots. That optimal value of the average effective speed, remains constant as the chunk length getting larger than the optimal value of the chunk length. The average effective speed has a horizontal asymptote at $\frac{2V_{mot}}{n+1}$

**Numerical values.** The values of the fixed system parameters are: $T_{com} = 10[ms]$, the physical speed $V_{mot} = 1[m/s]$. We consider that the time required by the Total Order Broadcast algorithm is: $T\ n$ where $T$ = 50[ms]. The values of the number of robots from one robot until 60 robots, and the chunk length varies from zero to 3 meters. The effective speed increases as the chunk length increases until it reaches a maximal value. Figure. 5.4 shows that, in a case of a system composed of 3 robots for example, the maximal average effective speed is $0.48[m/s]$ which corresponds to optimal chunk length $\approx 2[m]$.

### 5.4.4 Average effective speed vs number of robots

In this Subsection, we focus on the relation between the average effective speed $V$ with respect to the total number of robots $n$ in the system for a given value of the chunk length. The relation average effective speed vs number of robots is presented in Equation. 5.5. The effective speed decreases as the number of robots increases for a given chunk length, because a robot $R_i$ must wait for more robots.

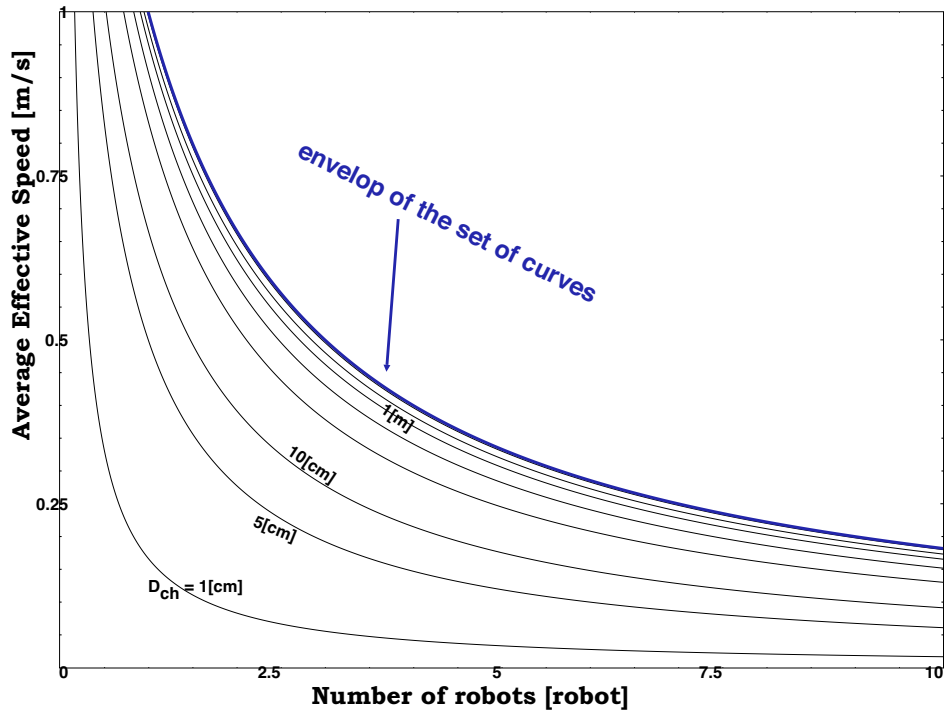



Figure 5.5: Average effective speed vs number of robots.

The derivative of the effective speed with respect to the number of robots is:

$$\frac{dV}{dn} = \frac{-D_{ch}(T + \frac{T_{com}}{2} + \frac{D_{ch}}{2V_{mot}})}{[Tn + \frac{n-1}{2}T_{com} + (\frac{n+1}{2V_{mot}})D_{ch}]^2} \tag{5.8}$$

Figure 5.5 shows the variation of the average effective speed with respect to the number of robots for different values of the chunk length.

**Numerical values.** The values of the fixed system parameters are: $T_{com} = 10[ms]$, the physical speed $V_{mot} = 1[m/s]$, $T = 50[ms]$. The values of the number of robots varies starting from a system with a single robot to a system with 10 robots, for different values of chunk length from $1[cm]$ to 10 meters. (Figure. 5.5).

The set of curves in Figure. 5.5 have an envelop curve, given by the following equation: $V = \frac{2V_{mot}}{n+1}$

- The envelop curve corresponds to the average effective speed for very high values of the chunk length (tends to infinity), since the average effective speed approaches to a constant value for a given number of robots in the system.

- All curves in Figure. 5.5 approaches to zero, when the number of robots tends to infinity. (horizontal asymptote at effective speed = 0).

## 5.5 Conclusion

In this chapter, we presented a fail-safe mobility management and achieved a collision prevention platform for asynchronous cooperative mobile robots in a closed group model.

Our fail-safe platform consists of a time-free collision prevention protocols, which guarantee that no collision can occur between robots. The collision prevention protocols are based on a distributed path reservation system. Each robot in the system knows the composition of the group, and can communicate with all robots of the group.

We have analyzed the performance of the protocol in terms of average effective speed of robots as a function of the chunk length and the number of robots. The effective speed depends also on some system parameters such as the average communication delays and the physical speed of robots. The performance analysis show that the average effective speed of robots increases with the chunk length for a given number of robots, and there is an optimal value of the chunk length that maximizes the average effective speed for a given number of robots. The performance analysis show also that the maximal value of the effective speed, remains constant while the chunk length is getting larger than the optimal value. The average effective speed decreases as the number of robots increases for a given chunk length. The effective speed of robots approaches to zero as the number of robots becomes very large.

This chapter presented collision prevention protocols for a closed group of mobile robots, and proved the correctness of the protocols and that they satisfy the properties of the collision prevention problem presented in Chapter 4 . The first protocol does not consider the crash of robots, while the two other protocols are fault-tolerant protocols designed for robotic systems prone to robot crashes.

The two fault-tolerant collision prevention protocols rely on $\diamond\mathcal{S}$ failure detector and tolerate the failures of half of the robots. One of the fault-tolerant collision prevention protocols is preemptive in the sense that a request of a robot can be preempted due to a decision voted by the other robots of the group, if the robot is suspected as a crashed robot. The second protocol is non-preemptive, so a robot $R_i$ cancels its own request when it suspects a robot $R_j$ as a crashed robot, if $R_i$ waits for $R_j$.

Both protocols tolerate the failures of half of the robots. The preemptive protocol reduces the negative drawback of wrong suspicions, since a robot is considered as a crashed robot if it is suspected by the majority of robots, while the non-preemptive protocol has the potential to do more wrong suspicions than the preemptive protocol, since a robot $R_j$ is considered as a crashed robot based on the suspicion of a robot $R_i$ individually, but the reaction against suspicion affects the robot $R_i$ if it has wrongly suspected $R_j$, so $R_i$ cancels its request and restarts a new request of an alternative zone. On the other hand, the decision of suspecting a robot $R_j$, in the preemptive protocol, affects the robot $R_j$, if it has not owned its requested zone $Zone_j$, so $R_j$ is obliged

to retry again and restarts its request of the same zone $Zone_j$. The preemptive and the non-preemptive protocols exhibit the same behavior if a suspected robot $R_j$ has owned its requested zone $Zone_j$, so a robot $R_i$ that waits for $R_j$ and $Zone_i$ intersects with $Zone_j$, must cancel its request and restarts a new request of an alternative zone, since $Zone_j$ is considered as a blocked zone until eventually $Zone_j$ is released if $R_j$ has not really crashed.

Both protocols ensure the *safety* property that no collision between robots can occur, based on a $\diamond\mathcal{S}$ failure detector with the majority of correct robots. However, the liveness property is ensured by the non-premtive protocol, while the preemptive protocol requires $\diamond\mathcal{P}$ failure detector in order to ensure the liveness property.

On the other hand, the non-preemptive protocol requires more trials to request an alternative zone by a robot $R_i$ if it suspects $R_j$, thus more exceptions are raised, while in the preemptive protocol, a robot $R_i$ requests an alternative zone when it conflicts with a robot $R_j$ that is suspected by the majority of robots and only if $R_j$ has owned its requested zone $Zone_j$.

# Chapter 6

# Locality-preserving collision prevention protocol for a dynamic group model

## 6.1 Dynamic group model

We consider a dynamic system of mobile robots S = $\{R_i\}$ in which each robot has a unique identifier. The total composition of the system, of which robots have only a partial knowledge, can change dynamically. Robots have access to a global positioning device that, when queried by a robot $R_i$, returns $R_i$'s position with a bounded error $\varepsilon_{gps}$. The robots communicate using wireless communication with a limited range $D_{tr}$. If the distance between two robots $R_i$ and $R_j$ is less than $D_{tr}$, then the two robots can communicate with each other. Communications assume retransmissions mechanisms such that communication channels are reliable. The system is asynchronous in the sense that there is no bound on communication delays, processing speed and on robots speed movement. Each robot has an access to a neighborhood discovery primitive named *NDiscover*.

**Neighborhood discovery (*NDiscover*)**

**Characteristics.** The neighborhood discovery primitive called *NDiscover* is a function that enables a robot to detect its local neighbors. These neighbors are within one communication hop and satisfy a certain known predefined condition.

**Implementation.** *NDiscover* can be implemented as the traditional neighborhood discovery primitive of mobile ad hoc networks. An implementation of *NDiscover* primitive can be performed by Geocasting[1] a ping message in a geographical region centered on the robot at the time of calling *NDiscover* with a radius within the transmission range. All the robots that

[1]Geocast is defined by the transmission of a message to a predefined geographical region

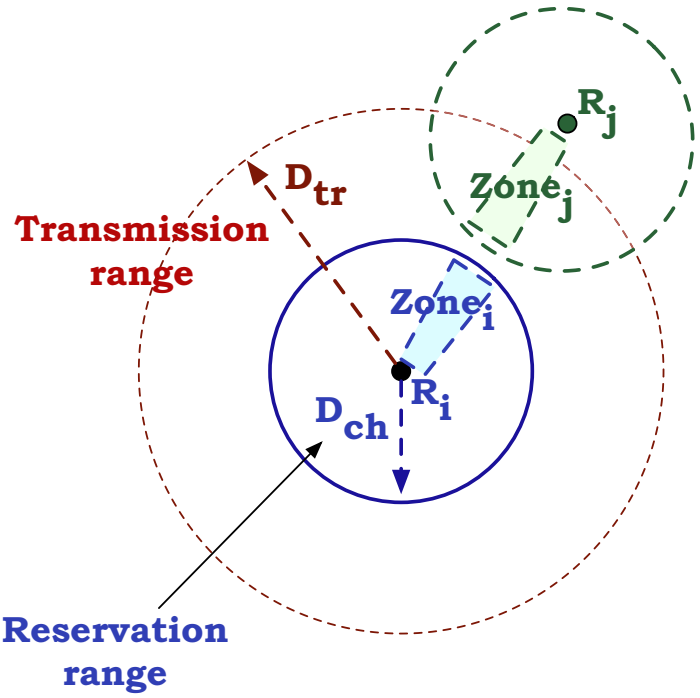



Figure 6.1: The reservation range is within half of the transmission range. Robot $R_i$ cannot communicate with robot $R_j$, and $Zone_i$ does not intersect with $Zone_j$.

receive the message and satisfy the predefined condition acknowledge the caller of *NDiscover*.[2]

**Reservation range** Robots have a limited wireless transmission range. It follows that a reserved zone by a robot must be entirely within a circle centered on the robot with a radius within half of the transmission range. The motivation behind this maximal value is that each robot can communicate with all the robots that it might collide with. Figure 6.1 illustrates the reservation range property.

The collision prevention protocol provides a parameter named *reservation range* and denoted $D_{ch}$, that is within half of the transmission range ($D_{ch} \leq \frac{D_{tr}}{2}$), such that a reserved zone by a robot is entirely within a circle centered on the robot with a radius equals to the *reservation range*.

## 6.2 Collision Prevention: locality-preserving protocol

All robots run the same distributed algorithm which is based on the following idea. When a robot $R_i$ requests a zone $Zone_i$, $R_i$ must determine all the robots $R_j$ that conflict with $R_i$. The robots $R_j$ are located within one communication hop with respect to $R_i$, because the reservation

[2] An implementation of *NDiscover* requires timing property for transmitting and processing the ping messages. This timely behavior can be particularly achieved, since *NDiscover* relies on very lightweight ping messages carrying only the position coordinates of the caller.

range of the robots must be within half of the transmission range. The Neighborhood Discovery primitive returns the set of neighbors $Neighbor_i$ within one communication hop with respect to $R_i$. Therefore, $R_i$ can determine the set of robots $R_j$ that conflict with $R_i$. $R_i$ multicasts $Zone_i$ to the list of neighbors $Neighbor_i$, then $R_i$ waits until receive the response messages. Consequently, $R_i$ determines the set of robots that it conflicts with. Intuitively, $R_i$ performs a pair-wise negotiation with each of the robots that $R_i$ conflicts with. Therefore, $R_i$ and each robot $R_j$ decide consistently about the scheduling of their requests. So, a dynamic scheduling for the conflicting requests takes place. When $R_i$ receives a release message from all the robots that $R_i$ waits for, it reserves $Zone_i$ and becomes the owner of $Zone_i$. After $R_i$ has reached the *post-motion* zone, $R_i$ releases $Zone_i$ except for the area occupied by $R_i$.

### 6.2.1 Variables

We present the variables used in the protocol.

- $Zone_i$ is the zone currently requested or owned by robot $R_i$.
- $Neighbor_i$ represents the set of robots that may possibly conflict with robot $R_i$ (i.e., the output of the neighborhood discovery primitive $NDiscover_i$).
- $G_i$ is a set of $\{(R_j, Zone_j)\}$ such that $R_j$ belongs to $Neighbor_i$, and $Zone_j$ is the requested or the owned zone of $R_j$ such that $Zone_j$ intersects with $Zone_i$.
- $WLAfter_i$ is the list of robots waiting for $R_i$ until it releases its zone.
- $WLBefore_i$ is the list of robots that $R_i$ waits for.
- $Depend_i$ is the *dependency* set. If a robot $R_i$ requests $Zone_i$ then it conflicts with a set of robots each of which conflicts with another set of robots and so on. The *dependency* set is the union of $G_k$ for each robot $R_k$ related to $R_i$ by the *transitive closure* of the relation *conflict*.
- *Dag* is a *wait-for* graph such that the vertices represent robots and a directed edge from $R_i$ to $R_j$ represent that $R_i$ waits for $R_j$ to release $Zone_j$.
- *msg* denotes a message exchanged during the run of the protocol. Each *msg* message consists of three fields, the first is the type of the message which belongs to the set {REQUEST, RELEASE, WAITFORME}, the second field is the identifier of the robot sending the message, and the third field is the body of the message which consists of the specifications and the parameters of the requested (or owned) zone. The type REQUEST denotes a request message, RELEASE denotes a release message, and the type

WAITFORME means that the receiver of the message must wait for the sender. (This *wait-for* relation is called an *imposed wait-for* relation).

### 6.2.2 Protocol description

We explain the phases of the protocol with respect to a robot $R_i$. The robot $R_i$ is located in the *pre-motion* zone $pre(Zone_i)$. When robot $R_i$ requests a new zone $Zone_i$, it proceeds as follows.

1. Discovery phase: $R_i$ calls the neighborhood discovery primitive $NDiscover_i$, to determine the set $Neighbor_i$. This set consists of robots $R_j$, that *may* possibly come in conflict with $R_i$ for $Zone_i$, since $Zone_j$ intersects with the circle centered on $R_i$ with radius equals to the *reservation range*.

2. Negotiation phase: The Negotiation phase of $R_i$ starts by the determination of the set $G_i$ which consists of the robots of $Neighbor_i$ that *conflict* with $R_i$. The output of the Negotiation phase is the *wait-for* graph, $Dag_{wait}$. Thus, $R_i$ determines the set of robots that it waits for. If $R_i$ receives a request from a robot $R_k$ ($Zone_k$ intersects with $Zone_i$) and $R_k$ does not belong to $G_i$, then $R_k$ must wait for $R_i$. The Negotiation phase proceeds as follows.

   - $R_i$ multicasts a message $msg_i = (\text{REQUEST}, i, Zone_i)$ indicating that $R_i$ requests $Zone_i$ to all the members of $Neighbor_i$ carrying the parameters of $Zone_i$. This multicast does not require any routing because the neighbors are located within one communication hop with respect to $R_i$.
   - $R_i$ waits until it receives a response message $msg_j$ from each member $R_j \in Neighbor_i$.
   - After $R_i$ has received the messages $msg_j$, $R_i$ determines the set of robots $G_i$ that *conflict* with $R_i$. ($G_i$ is obtained from the received messages $msg_j$ after discarding the release messages $msg_j = (\text{RELEASE}, j, Zone_j)$, and discarding also the request messages $msg_j = (\text{REQUEST}, j, Zone_j)$ such that $Zone_j$ does not intersect with $Zone_i$). The set $G_i$ contains two disjoint subsets of robots: the first subset denoted $(G1)_i$ is composed of robots $R_j$ such that $R_i$ does not belong to $G_j$ (i.e., $R_i$ must wait for $R_j$, $R_j$ has sent the message $msg_j = (\text{WAITFORME}, j, Zone_j)$). The second is the complementary subset denoted $(G2)_i$, which is composed of robots $R_j$ such that $R_i$ belongs to $G_j$. Thus $R_i$ must wait for all the robots of $(G1)_i$, in addition to some robots of $(G2)_i$. (These robots would be determined later).
   - $R_i$ determines the *dependency* set $Depend_i$ by applying an *Echo* algorithm inspired from [28]. The *Echo* algorithm is explained as follows. $R_i$ multicasts a token message to each robot that belongs to $G_i$. Upon receipt of the first message of $R_i$ by a

robot $R_k$ from $R_j$ ($R_j$ is called the *father* of $R_k$), it multicasts the message of $R_i$ to all the robots of $G_k$ except its *father* $R_j$. When a robot $R_k$ has received the token message of $R_i$ from all the robots of $G_k$, $R_k$ adds the contents of $G_k$ to the token message and sends it (*echo*) to the *father* $R_j$. When $R_i$ has received the token message from all the robots of $G_i$, it obtains the *dependency* set[3]. The motivation for building the dependency set is to enable the conflicting robots to build the *wait-for* graph $Dag_{wait}$ in a consistent manner and so to avoid cyclic wait-for relations.

- $R_i$ uses the dependency set $Depend_i$ to construct $Dag_{wait}$. The vertices represent the robots of the set $Depend_i$ and a directed edge from $R_i$ to $R_j$ means that $R_i$ waits for $R_j$. $Dag_{wait}$ is built as follows. $R_i$ starts by establishing the *imposed wait-for* relations (Subsection. 6.2.3), and then it breaks ties for the remainder of the conflicting robots by applying a specified conflict resolution *policy*. At first, $R_i$ builds a graph named WAITFORME graph and denoted $Dag_{wm}$. This graph corresponds to the relation between $R_i$ and $R_j$ from the set $(G1)_i$. (Subsection 6.2.3). The next step, $R_i$ builds a graph named $Dag_{dr}$ by adding the directed edges imposed by the *deadlock risk* situations. (Subsection 6.2.3). After having established the *imposed wait-for* relations, $R_i$ adds the directed edges that result from resolving the conflicts according to a specified policy. (Subsection 6.2.4). $R_i$ and the conflicting robots build the same directed acyclic graph $Dag_{wait}$ in a consistent manner.
- According to the graph $Dag_{wait}$, $R_i$ determines $WLBefore_i$ the set of robots that $R_i$ waits for. ($WLBefore_i$ = $(G1)_i$ and some robots of $(G2)_i$). $R_i$ updates the set $WLAfter_i$ of robots that wait for $R_i$, due to the graph $Dag_{wait}$. $R_i$ dynamically updates the set $WLAfter_i$ by adding robots $R_k$ that does not belong to $G_i$ and whose requested zone $Zone_k$ intersects with $Zone_i$. ($R_i$ sends to $R_k$ the message $msg_i$ = (WAITFORME, $i$, $Zone_i$)). $R_i$ keeps updating the set $WLAfter_i$ until $R_i$ releases the zone.
- $R_i$ waits until receives a release message from each robot in the set $WLBefore_i$.

3. Reservation phase: When $R_i$ has received a release message from all the robots of the set $WLBefore_i$, or (when the set $WLBefore_i$ is empty), $R_i$ reserves $Zone_i$ and becomes the owner of $Zone_i$.

4. Release phase: When $R_i$ reaches the *post-motion* zone $post(Zone_i)$, it releases $Zone_i$ except the place that $R_i$ occupies. $R_i$ multicasts a release message to all the robots that belong to the set $WLAfter_i$ and to robots $R_a$ such that $R_i \in Neighbor_a$ (due to $NDiscover_a$

[3]The *dependency* set is not computed each time $R_i$ requests a zone. It can be "piggybacked" with the messages of type WAITFORME.

primitive) and the request message $msg_a$ = (REQUEST, a, $Zone_a$) of $R_a$ is not received yet by $R_i$. When $R_i$ reaches the *post-motion* zone, the robots of the set $WLAfter_i$ and the robots $R_a$ are within one communication hop with respect to $R_i$, hence the robots of the set $WLAfter_i$ and the robots $R_a$ can receive the release message of $R_i$.

### 6.2.3 Imposed wait-for relations

If a robot $R_i$ conflicts with robot $R_j$, then the Conflict Resolver determines the *wait-for* relation by breaking ties between $R_i$ and $R_j$, according to a specified *policy*. However, there are situations where the *wait-for* relation is imposed. The situations where the *wait-for* relation is imposed are discussed in the WAITFORME Handler and the Deadlock detector.

**WAITFORME Handler** The input of the WAITFORME Handler is the dependency set $Depend_i$, and the output is the directed acyclic graph $Dag_{wm}$. This handler generates $Dag_{wm}$ by establishing the imposed *wait-for* directed edges that correspond to the situation where $R_i$ must wait for $R_j$ because $R_j$ is a member of the set $(G1)_i$. (Subsection. 6.2.2). The relation WAITFORME is transitive, so if a robot $R_i$ must wait for $R_j$ and the robot $R_j$ must wait for $R_k$, then $R_i$ must wait for $R_k$. Therefore, no cycles can be created in the graph $Dag_{wm}$.

**Deadlock detector** There are specified intersection situations between $Zone_i$ and $Zone_j$, such that neither $R_i$ nor $R_j$ can move, because if $R_i$ is granted $Zone_i$ and eventually released $RelZone_i$ before $R_j$ is granted $Zone_j$ then, a collision may occur between $R_i$ and $R_j$. Also, if $R_j$ is granted $Zone_j$ and eventually released $RelZone_j$ before $R_i$ is granted $Zone_i$ then, a collision may occur between $R_i$ and $R_j$. We say that $R_i$ and $R_j$ are in a deadlock situation because none of the robots can own its requested zone. The deadlock situations are discussed in details in Chapter 3.

The *deadlock risk* situation imposes *wait-for* relations between two conflicting robots. If $R_i$ is in *deadlock risk*$_{pre}$ situation with $R_j$, then $R_i$ must wait for $R_j$. So that, $R_j$ releases $pre(Zone_j)$ before $R_i$ owns $Zone_i$. If $R_i$ is in *deadlock risk*$_{post}$ situation with $R_j$, then $R_j$ must wait for $R_i$.

The deadlock detector establishes the *wait-for* relations between the conflicting robots of the dependency set $Depend_i$ according to the *deadlock risk* situations. The deadlock detector adds the directed edges imposed by the *deadlock risk* situations, to the graph $Dag_{wm}$ generated by the WAITFORME Handler. If a cycle is created by adding a directed edge to the graph $Dag_{dr}$, then the deadlock detector calls the Deadlock Handler. A cycle in $Dag_{dr}$ results from the following situation. A robot $R_i$ must wait for a robot $R_j$ (WAITFORME relation) and $Zone_i$ intersects with $post(Zone_j)$. Thus, the deadlock detector generates the directed acyclic graph $Dag_{dr}$ or calls the Deadlock Handler in case of detecting a deadlock situation or a cycle.

**Algorithm 15** Collision prevention protocol (Code for robot $R_i$)

1: **Initialization:** $G_i := \emptyset$; $WLBefore_i := \emptyset$; $WLAfter_i := \emptyset$;

2: **procedure** Request(Zone)
3: **Phase 1:**
4: $NDiscover_i \Rightarrow Neighbor_i$ {*Neighborhood discovery*}

5: **task** Reply
6: **when** reception of a message $msg_i = (\text{REQUEST}, k, Zone_k)$
7: **if** $R_k \notin Neighbor_i$ and $Zone_k$ intersects with $Zone_i$ **then**
8: Send($msg_i = (\text{WAITFORME}, i, Zone_i)$) to $R_k$ {*The task* Reply *runs in parallel with the next phases, until* $R_i$ *releases the zone*}
9: $WLAfter_i := WLAfter_i \cup \{R_k\}$ {$R_i$ *keeps updating the set* $WLAfter_i$ *until* $R_i$ *releases the zone*}
10: **end if**
11: **end when**
12: **end**

13: **Phase 2:**
14: multicast ($msg_i = (\text{REQUEST}, i, Zone_i)$) to $Neighbor_i$ {*Negotiation*}
15: **wait until** receive response $\text{msg}_j$ from all $R_j \in Neighbor_i$
16: build the set $G_i = (R_j, Zone_j)$ such that $R_j \in Neighbor_i$ and $Zone_j$ intersects with $Zone_i$.
17: determine the *dependency* set $Depend_i$
18: $Dag_{wm}$ := WAITFORME Handler($Depend_i$)
19: $Dag_{dr}$ := Deadlock Detector($Dag_{wm}$, $Depend_i$)
20: $Dag_{wait}$ := Conflict Resolver($Dag_{dr}$, $Depend_i$, policy)
21: build the set $WLBefore_i$ and update the set $WLAfter_i$ according to the directed acyclic graph $Dag_{wait}$
22: **if** $WLBefore_i \neq \emptyset$ **then**
23: **while** reception of a release message from $R_j \in WLBefore_i$ **do**
24: $G_i := G_i \setminus \{R_j, Zone_j\}$ {$R_i$ *removes the entry of* $R_j$ *from the set* $G_i$}
25: **end while** {*receive the release message from all* $R_j$ *of the set* $WLBefore_i$}
26: **end if**
27: **end** Request

28: **Phase 3:**
29: reserve($R_i$, $Zone_i$) {$R_i$ *reserves the zone* $Zone_i$}

30: **procedure** Release(Zone)
31: **Phase 4:**
32: **when** $R_i$ reaches the *post-motion* zone *post*($Zone_i$)
33: **if** $WLAfter_i \neq \emptyset$ **then**
34: multicast($RelZone_i$) to $WLAfter_i$ {*release*($R_i$, $RelZone_i$)}
{$R_i$ *multicasts a release message to all* $R_j$ *of the set* $WLAfter_i$}
35: **end if**
36: **end when**
37: **end** Release

**Algorithm 16** WaitForMe Handler algorithm

```
1:  function WaitForMe-Handler (Depend_i)
2:     for all (R_x, R_y) ∈ Depend_i do
3:        if R_x must wait for R_y (WAITFORME) then
4:           Dag_wm := Dag_wm ∪ DirEdge(R_x, R_y)        {R_x must wait for R_y, because R_y ∈ G_x but R_x ∉ G_y}
5:        end if
6:        if DirEdge(R_x, R_y) and DirEdge(R_y, R_z) then
7:           Dag_wm := Dag_wm ∪ DirEdge(R_x, R_z)        {The relation WAITFORME is transitive}
8:        end if
9:     end for
10:    return Dag_wm
11: end
```

**Algorithm 17** Deadlock detector algorithm

```
1:  function Deadlock-Detector (Dag_wm, Depend_i)
2:     Dag_dr := Dag_wm
3:     for all (R_x, R_y) ∈ Depend_i do
4:        Deadlock situation 1:=
          [Zone_x ∩ pre(Zone_y) ≠ ∅] and [Zone_y ∩ pre(Zone_x) ≠ ∅] and [post(Zone_x) ∩ post(Zone_y) = ∅]
5:        Deadlock situation 2:=
          [Zone_x ∩ post(Zone_y) ≠ ∅] and [Zone_y ∩ post(Zone_x) ≠ ∅] and [post(Zone_x) ∩ post(Zone_y) = ∅]
6:        Deadlock situation 3:=
          [Zone_x ∩ pre(Zone_y) ≠ ∅] and [Zone_x ∩ post(Zone_y) ≠ ∅] and [post(Zone_x) ∩ post(Zone_y) = ∅]
7:        Deadlock situation 4:=
          post(Zone_x) ∩ post(Zone_y) ≠ ∅

8:        if Deadlock situation 1 or Deadlock situation 2 or Deadlock situation 3 or Deadlock situation 4 then
9:           Deadlock Handler(Deadlock situation)
10:       end if
11:       if deadlock risk_pre(R_x, R_y) or deadlock risk_post(R_y, R_x) then
12:          Dag_dr := Dag_dr ∪ DirEdge(R_x, R_y)        {R_x waits-for R_y}
13:       end if
14:       if deadlock risk_post(R_x, R_y) or deadlock risk_pre(R_y, R_x) then
15:          Dag_dr := Dag_dr ∪ DirEdge(R_y, R_x)        {R_y waits-for R_x}
16:       end if
17:       if DetectCycle then
18:          Deadlock Handler(Deadlock situation)
19:       end if
20:    end for
21:    return Dag_dr
22: end
```

### 6.2.4 Conflict Resolver

The *Conflict Resolver* breaks ties and determines the *wait-for* relation between two conflicting robots according to a conflict resolution *policy*, if there is no imposed *wait-for* relation between the two robots. A conflict resolution policy can be as follows. $R_i$ *waits-for* $R_j$ if the number of the previous requested zones by $R_i$ is higher than that of $R_j$. In our protocol, the conflict resolution policy is specified by the robotic application. For example, the robot farther to the intersection zone *waits-for* the closer one, and in case of an equidistance situation, their identifiers are used to break the symmetry. The Conflict Resolver generates the graph $Dag_{wait}$ by breaking ties between each pair of the robots of the dependency set $Depend_i$. The graph $Dag_{wait}$ is generated in a consistent manner, such that each robot of the set $Depend_i$ generates the *same* graph $Dag_{wait}$ starting from the graph $Dag_{dr}$ by adding the directed edges representing the *wait-for* relations after resolving the conflict between each pair of the conflicting robots. The dependency set is scanned according to the increasing order of the identifiers of robots and the conflict resolution policy is applied. If adding a directed edge creates a cycle then the new directed edge is reversed to break the cycle. A robot $R_i$ determines the set of robots that it waits for $WLBefore_i$, and updates the set $WLAfter_i$ according to the directed acyclic graph $Dag_{wait}$.

**Algorithm 18** Conflict Resolver algorithm

```
 1: function Conflict-Resolver (Dag_dr, Depend_i, policy)
 2:     Dag_wait := Dag_dr
 3:     for each robot's identifier x from MINID to MAXID such that R_x ∈ Depend_i do
 4:         for each robot's identifier y > x to MAXID such that R_y ∈ Depend_i do
 5:             if Conflict(R_x, R_y) and no edge (R_x, R_y) then
 6:                 DirEdge(R_x, R_y) := policy(R_x, R_y)        {apply the conflict resolution policy}
 7:                 Dag_wait := Dag_wait ∪ DirEdge(R_x, R_y)
 8:                 if DetectCycle then
 9:                     DirEdge(R_x, R_y) := DirEdge(R_y, R_x) {If a cycle is detected then inverse the direction of the edge}
10:                 end if
11:             end if
12:         end for
13:     end for
14:     return Dag_wait
15: end
```

### 6.2.5 Deadlock Handler

The Deadlock Handler resolves a deadlock situation detected by the deadlock detector module. The policy used by the Deadlock Handler to resolve a deadlock situation is based on a *Request Preemption* strategy. Hence, if there is a deadlock situation between two requests (Deadlock situation 1, Deadlock situation 2, or Deadlock situation 4) then, the request which has the larger robot's identifier is preempted (the set $Depend_i$ is scanned according to the increasing order of robots identifiers).

If a robot $R_i$ must wait for a robot $R_j$ (WAITFORME relation) and $Zone_i$ intersects with $post(Zone_j)$ then, the request ($R_i$, $Zone_i$) is preempted. This situation is denoted by "Cycle-Situation" in Algorithm 19.

If Deadlock situation 3 ($Zone_i$ intersects with both $pre(Zone_j)$ and $post(Zone_j)$) then, the request ($R_i$, $Zone_i$) is preempted.

When a request is preempted, the sets $G_i$ and $WLAfter_i$ are updated by removing the preempted request.

The Deadlock Handler proposes an alternative chunk to the robot $R_k$ which request was preempted. $R_k$ restarts a new request of an alternative zone. The Deadlock Handler of a robot $R_k$ proposes an alternative chunk offered by the motion planning layer. If there is no possible alternative chunk then, the Deadlock Handler raises an exception.

The design of the collision prevention protocol yields a flexibility to handle the exceptions caused by deadlock situations, due to the module Deadlock Handler. The Deadlock Handler can apply an application-based policy in order to resolve deadlock situations.

**Algorithm 19** Deadlock Handler algorithm

```
1:  function Deadlock Handler (R_i, Zone_i, R_j, Zone_j)

2:     Cycle-Sitaution := Zone_i intersects with post(Zone_j) and R_i must wait for R_j

3:     if Deadlock situation 3 or Cycle-Situation then
4:        Request Preemption (R_i, Zone_i)
5:     else
6:        Request Preemption (the request of the higher robot's identifier)
                                  {Deadlock situation 1, Deadlock situation 2, or Deadlock situation 4}
                                  {Preempt the request which has the higher robot's identifier}
7:     end if

8:     if no possible alternative chunk then
9:        throw Exception                                          {There is no solution}
10:       return Exception
11:    end if
12:    Zone := Zone^alternative
13:    Request (Zone)                    {The Deadlock Handler proposes an alternative chunk}
14: end
```

## 6.3 Example

Consider the following example illustrated in Figure 6.2. A robot $R_i$ requests $Zone_i$. The neighborhood discovery $NDiscover_i$ returns the set $Neighbor_i = \{R_a, R_b, R_j, R_k\}$ since each of $Zone_a$, $Zone_b$, $Zone_j$, $Zone_k$ intersects with the circle centered on $R_i$ with radius $D_{ch}$.

$R_i$ multicasts the parameters of $Zone_i$ to $\{R_a, R_b, R_j, R_k\}$. Then $R_i$ waits until receive the messages $\{msg_a, msg_b, msg_j, msg_k\}$.

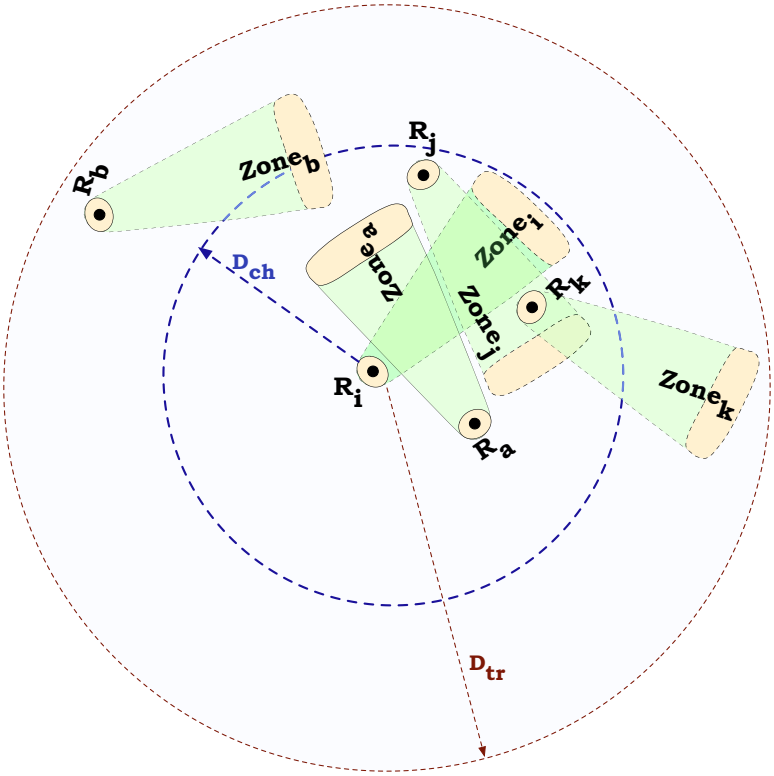
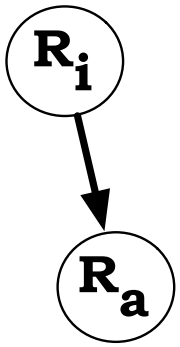
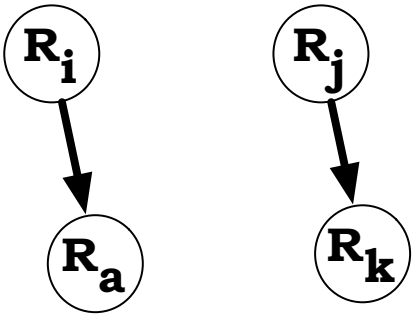



Figure 6.2: Example. $R_i$ requests $Zone_i$ and $Neighbor_i = \{R_a, R_b, R_j, R_k\}$.




(a) The graph $Dag_{wm}$ (WAITFORME relations).




(b) The graph $Dag_{dr}$ (Deadlock risk imposed wait-for relations).

Figure 6.3: The graphs $Dag_{wm}$ and $Dag_{dr}$ related to the imposed wait-for relations.

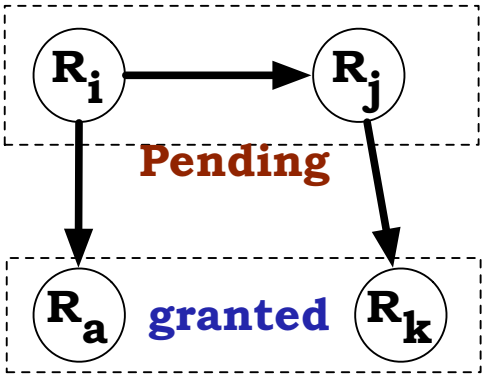



Figure 6.4: The wait-for graph $Dag_{wait}$.

- $R_i$ discards $msg_b$ because $Zone_b$ does not intersect with $Zone_i$.
- $R_i$ discards $msg_k$ because $Zone_k$ does not intersect with $Zone_i$.
- $R_i \notin G_a$, so $msg_a$ = (WAITFORME, a, $Zone_a$).
- $R_i \in G_j$, so $msg_j$ = (REQUEST, j, $Zone_j$).

So, $G_i = \{(R_a, Zone_a), (R_j, Zone_j)\}$. $(G1)_i = \{(R_a, Zone_a)\}$, $(G2)_i = \{(R_j, Zone_j)\}$. The dependency set $Depend_i = \{(R_a, Zone_a), (R_j, Zone_j), (R_k, Zone_k)\}$.

The graph $Dag_{wm}$ related to the WAITFORME relations is presented in Figure 6.3(a). $Zone_j$ intersects with $pre(Zone_k)$ hence, $R_j$ must wait for $R_k$, (Deadlock $risk_{pre}(R_j, R_k)$, and also Deadlock $risk_{post}(R_k, R_j)$). The graph $Dag_{dr}$ related to the Deadlock risk imposed wait-for relations is presented in Figure 6.3(b).

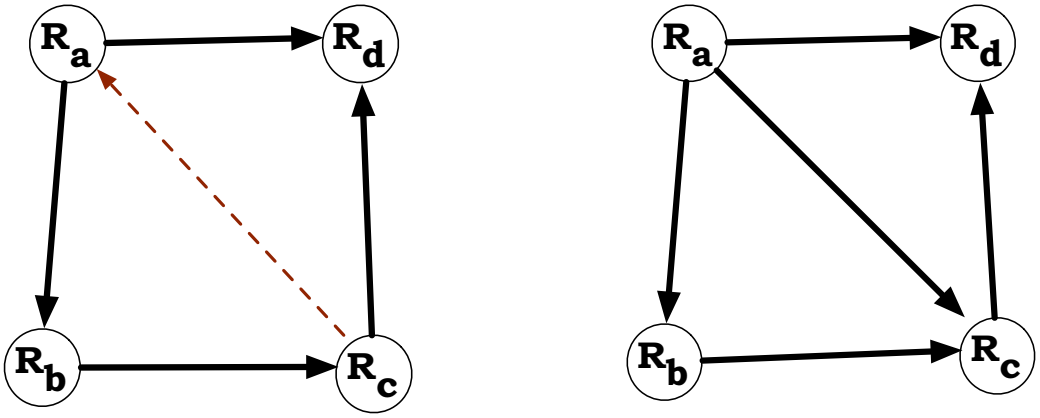
The wait-for graph $Dag_{wait}$ is generated by adding the directed edge $(R_i, R_j)$ to the graph $Dag_{dr}$. The Conflict Resolution policy determines the wait-for relation between $R_i$ and $R_j$. The wait-for graph $Dag_{wait}$ is illustrated in Figure 6.4.

## 6.4 Proof of correctness

We prove that the collision prevention protocol for a dynamic group model satisfies the following properties. The *Safety* property, the *Liveness* property and the *Non Triviality* property presented in Chapter 4.

**Lemma 8** *The wait-for graph $Dag_{wait}$ has no cycles.*

**Proof.** The wait-for graph $Dag_{wait}$ is based on $Dag_{wm}$ and $Dag_{dr}$.




(a) A directed edge ($R_c$, $R_a$) is added.

(b) The direction is reversed, so the directed edge is replaced by ($R_a$, $R_c$).

Figure 6.5: Adding a directed edge to the wait-for graph $Dag_{wait}$.

- The graph $Dag_{wm}$ is a directed acyclic graph, since the WAITFORME relation is transitive. if a robot $R_x$ must wait for $R_y$ and the robot $R_y$ must wait for $R_z$, then $R_x$ must wait for $R_z$. (Algorithm 16, Line 7). Therefore, no cycles can be created in the graph $Dag_{wm}$.

- The graph $Dag_{dr}$ is a directed acyclic graph. If a robot $R_i$ must wait for a robot $R_j$ (WAITFORME relation) and $Zone_i$ intersects with $post(Zone_j)$ then, the Deadlock detector algorithm detects a cycle, (Algorithm 17, Line 18) and consequently calls the Deadlock Handler. Therefore, if a cycle is detected then, the Deadlock Handler breaks the cycle by preempting the request ($R_i$, $Zone_i$). Therefore, the graph $Dag_{dr}$ does not contain a cycle.

- We prove that the graph $Dag_{wait}$ is a directed acyclic graph. $Dag_{wait}$ is generated starting from $Dag_{dr}$ which is a directed acyclic graph. If adding a directed edge to $Dag_{wait}$ creates a cycle, then the direction is reversed. We prove that reversing the direction of the edge does not create a cycle and hence the wait-for graph $Dag_{wait}$ is a directed acyclic graph.

  For a directed acyclic graph that consists of three vertices $\{R_a, R_b, R_c\}$, if adding the directed edge ($R_c$, $R_a$) creates a cycle then, the direction is reversed, and the edge ($R_a$, $R_c$) obviously breaks the cycle.

  For a directed acyclic graph that consists of more than three vertices, the proof proceeds by contradiction.

  Let us assume that the directed edge ($R_a$, $R_c$) creates a cycle. So, the vertex $R_d$ participates in the created cycle, thus there is a path from $R_c$ to $R_a$ via $R_d$. But, by assumptions the edge ($R_c$, $R_a$) creates a cycle via $R_b$. Consequently, there exists a cycle ($R_a$, $R_b$, $R_c$, $R_d$) and the original graph is not a directed acyclic graph, which leads to a contradiction. Therefore, reversing the direction of an edge does not create a cycle in $Dag_{wait}$.

Figure 6.5(a) shows that the directed edge ($R_c$, $R_a$) is added to $Dag_{wait}$, so it creates the cycle ($R_a$, $R_b$, $R_c$). Figure 6.5(b) shows that the edge is replaced by ($R_a$, $R_c$) and $Dag_{wait}$ is a directed acyclic graph.

Therefore, the wait-for graph $Dag_{wait}$ has no cycles. □

**Lemma 9** *The wait-for relations between robots related by the transitive closure of the relation (conflict), are generated consistently, so the robots build the same wait-for graph* $Dag_{wait}$.

**Proof.** The set $Depend_i$ consists of the union of $G_k$ for each robot $R_k$ related to $R_i$ by the transitive closure of the *conflict* relation, so $Depend_i$ equals to $Depend_k$.

The robots that $R_i$ conflicts with, belong to $G_i$ or to $WLAfter_i$. We prove that the set $G_i$ is sufficient to build the wait-for graph $Dag_{wait}$ consistently.

Let us consider three conflicting robots $R_a$, $R_b$ and $R_c$, such that each zone intersects with the two other zones. The sets $G_a$, $G_b$ are as follows. the set $G_a$ contains $R_b$, but does not contain $R_c$, ($R_c \in WLAfter_a$). The set $G_b$ contains both $R_a$ and $R_c$. When the dependency set $Depend_b$ is computed, $R_b$ deduces the wait-for relation between $R_a$ and $R_c$ and that $R_c$ waits-for $R_a$, since the $Zone_a$ intersects with $Zone_c$ and $R_c \notin G_a$.

If $R_a$ receives the set $G_b$ (due to the dependency set $Depend_a$) before $R_a$ receives the request message of $R_c$ then, $R_a$ deduces that a request message of $R_c$ eventually arrives, and $R_c$ eventually belongs to the set $WLAfter_a$, since $Zone_a$ intersects with $Zone_c$.

If $R_c$ neither belongs to $G_a$ nor to $G_b$ then, $R_c$ belongs to $WLAfter_a$ and to $WLAfter_b$, so $R_c$ waits for $R_a$ and $R_b$.

We prove that the wait-for graph $Dag_{wait}$ is generated consistently. The wait-for graph $Dag_{wait}$ is generated by a robot $R_i$ based on the set $Depend_i$, by applying a sequence of deterministic functions. The graph $Dag_{wm}$ is generated according to the imposed wait-for relation WAITFORME. Then, the graph $Dag_{dr}$ is generated according to the imposed wait-for relations of the Deadlock risk$_{pre}$ and Deadlock risk$_{post}$ situations (Lemmas [1, 2], Chapter 5). Since, the imposed wait-for relations define deterministic functions, then the wait-for graph $Dag_{dr}$ is generated consistently.

The Conflict Resolver defines a deterministic function (*policy*) to break ties between two conflicting robots, based on the graph $Dag_{dr}$ and the set $Depend_i$ which is scanned according to the increasing order of robots identifiers. Hence the wait-for graph $Dag_{wait}$ is generated consistently, so the robots that are related by the transitive closure of the relation (conflict) build the same wait-for graph. □

**Theorem 10 ( Mutual Exclusion)** *If a requested zone* $Zone_i$ *of* $R_i$ *intersects with a requested zone* $Zone_j$ *of* $R_j$ *then exclusively either* $R_i$ *or* $R_j$ *becomes the owner of its requested zone.*

$(Zone_i \cap Zone_j \neq \emptyset) \Rightarrow (R_i \text{ owns } Zone_i)$ XOR $(R_j \text{ owns } Zone_j)$

**Proof.** A robot $R_i$ requests $Zone_i$. If $Zone_i$ intersects with a zone $Zone_j$ of a robot $R_j$, then $R_j$ must be within the transmission range of $R_i$, (reservation range property: $D_{ch} \leq$ half of the transmission range), thus $R_i$ and $R_j$ can communicate.

The neighborhood discovery primitive $NDiscover_i$ returns $R_j \in$ the set of neighbors $Neighbor_i$, since $Zone_j$ intersects with the reservation zone of $R_i$, (the reservation zone of $R_i$ is the circle centered on $R_i$ with radius $D_{ch}$). Thus, $R_j \in G_i$.

- If $R_i \notin G_j$ then, $R_i$ must wait for $R_j$ (WAITFORME relation). If $Zone_i$ intersects with $post(Zone_j)$ then, this situation is detected by the Deadlock detector and the Deadlock Handler preempts the request ($R_i$, $Zone_i$) (Algorithm 19, Line 4).

- If $R_i \in G_j$ and there is no deadlock situation between $Zone_i$ and $Zone_j$ then, the wait-for relation is determined either by the Deadlock Detector (Algorithm 17) if there exists deadlock $risk_{pre}$(deadlock $risk_{post}$) imposed wait-for relation, or by the conflict resolver (Algorithm 18) otherwise. If there is a deadlock situation between $Zone_i$ and $Zone_j$ then, one of the requests is deterministically selected to be preempted.

Consequently, there is a wait-for relation between $R_i$ and $R_j$. According to Lemma 9 the wait-for relations between conflicting robots are generated consistently, so $R_i$ and $R_j$ establish the same wait-for relation and either $R_i$ waits for $R_j$ or $R_j$ waits for $R_i$.

Let us consider that $R_i$ waits for $R_j$, so $R_j$ releases $RelZone_j$, after that $R_i$ owns $Zone_i$. When the robot $R_i$ is the owner of $Zone_i$, the robot $R_j$ is deprived from its ownership to the zone $Zone_j$. The robot $R_j$ just keeps a part of $post(Zone_j)$ under its reservation. $Zone_i$ does not intersect with the part of $post(Zone_j)$ that is still owned by $R_j$, because:

1. $pre(Zone_i) \cap post(Zone_j) = \emptyset$ (Proof by contradiction). If $pre(Zone_i)$ intersects with $post(Zone_j)$, then this situation is the Deadlock $risk_{pre}(R_j, R_i)$ or the Deadlock $risk_{post}(R_i, R_j)$ situations. In both situations $R_j$ must wait for $R_i$ according to Lemmas. [1, 2], which leads to a contradiction, since the assumption is that $R_i$ waits for $R_j$.

2. $motion(Zone_i) \cap post(Zone_j) = \emptyset$ (Proof by contradiction). If the *motion* zone of $R_i$ intersects with the *post-motion* zone of $R_j$, then the situation is: Deadlock $risk_{post}(R_i, R_j)$. Thus, $R_j$ must wait for $R_i$ which leads to a contradiction.

3. $post(Zone_i) \cap post(Zone_j) = \emptyset$ (Proof by contradiction). If the *post-motion* zones intersect, then the situation is a deadlock situation (Deadlock situation 4), which leads to a contradiction.

Consequently, the ownership of intersecting zones satisfies the mutual exclusion property, and the *Safety* property holds. □

**Lemma 10** *The possible deadlock situations are as follows. Deadlock situations that belong to the set DS = {Deadlock situation 1, Deadlock situation 2, Deadlock situation 3, Deadlock situation 4}, and a situation where a robot $R_i$ must wait for a robot $R_j$, and $Zone_i$ intersects with post($Zone_j$).*

**Proof.** The same proof as for Lemma 3 in Chapter 5. □

**Theorem 11 (Non triviality)** *An exception is raised only when a deadlock situation occurs.*

**Proof.** An exception is raised by the Deadlock Handler (Algorithm 19, Line 9) when the Deadlock Handler does not find a solution for a deadlock situation.

The Deadlock Handler is called by the Deadlock Detector (Algorithm 17, Lines [9, 18]). Line 9 corresponds to a deadlock situation that belongs to the set *DS* = {Deadlock situation 1, Deadlock situation 2, Deadlock situation 3, Deadlock situation 4}. Line 18 corresponds to the deadlock situation, where a robot $R_i$ must wait for a robot $R_j$, and $Zone_i$ intersects with *post*($Zone_j$).

Therefore, an exception is raised only when a deadlock situation occurs. □

**Theorem 12 (Liveness)** *If a robot $R_i$ requests $Zone_i$ then eventually ($R_i$ owns $Zone_i$ or an exception is returned).*

$R_i$ requests $Zone_i \Rightarrow \Diamond$ ($R_i$ owns $Zone_i$ or Exception)

**Proof.** If a robot $R_i$ requests a zone $Zone_i$, then:

1. If $Zone_i$ does not intersect with a zone $Zone_j$, then $R_i$ owns $Zone_i$.
2. If $Zone_i$ intersects with a zone $Zone_j$, then a wait-for relation is established between $R_i$ and $R_j$ and a directed edge is added to the wait-for graph $Dag_{wait}$. According to Lemma. 8 the graph $Dag_{wait}$ has no cycles. Therefore, $R_i$ eventually owns $Zone_i$.
3. If a deadlock situation is detected, then the Deadlock Handler is called. If the Deadlock Handler algorithm does not find a solution to resolve a deadlock situation, then an exception is raised by the Deadlock Handler according to Theorem 11

4. Robot $R_j \in WLAfter_i$ eventually receives the release message of $R_i$ when $R_i$ reaches $post(Zone_i)$. Because, $R_i$ multicasts a release message to all the robots that belong to the set $WLAfter_i$ and to robots $R_a$ such that $R_i \in Neighbor_a$ (due to $NDiscover_a$ primitive) and the request message $msg_a$ = (REQUEST, a, $Zone_a$) of $R_a$ is not yet received by $R_i$. When $R_i$ reaches the *post-motion* zone, the robots of the set $WLAfter_i$ and the robots $R_a$ are within one communication hop with respect to $R_i$ (due to the reservation range property, $D_{ch} \leq$ half of the transmission range). Hence the robots of the set $WLAfter_i$ and the robots $R_a$ can receive the release message of $R_i$.

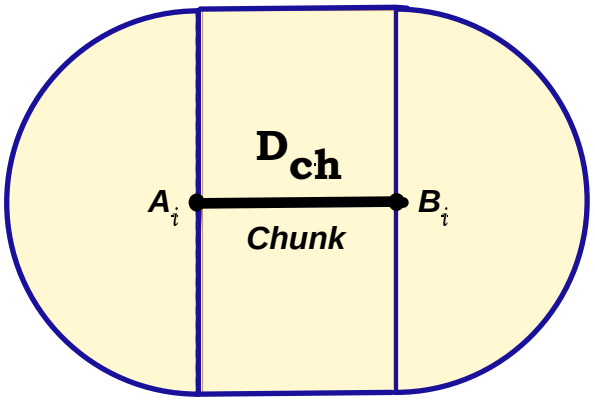
Therefore, $R_i$ requests $Zone_i \Rightarrow \Diamond$ ($R_i$ owns $Zone_i$ or Exception). □

## 6.5 Performance analysis

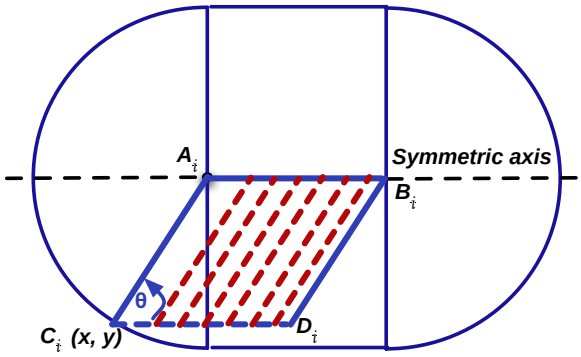
We study the performance of our protocol in terms of the time needed by a robot $R_i$ to reach a given destination when robots are active (robots do not sleep). We compute the average effective speed of robots executing our collision prevention protocol. We provide insights for a proper dimensioning of system's parameters in order to maximize the average effective speed of the robots. For simplicity, we assume in this section that the physical dimensions of robots are too small such that a robot can be considered as a point in the plane. The geometrical incertitude related to the positioning system, translational and rotational movement are neglected.

### 6.5.1 Intersection probability

Consider a set of robots, each one moves along a chunk (line segment) of length equal to the reservation range $D_{ch}$. At first let us determine the region *reg* around a line segment $A_iB_i$ of length $D_{ch}$ such that line segments of length $D_{ch}$ issued from points located in the region *reg may* possibly intersect with $A_iB_i$ and line segments of length $D_{ch}$ issued from points outside *reg can not* intersect with $A_iB_i$. The region *reg* is illustrated in Figure. 6.6(a). It represents the region of possible collisions for a robot $R_i$ that is located at point $A_i$ and moves along the line segment $A_iB_i$ of length $D_{ch}$. Consider the robots located in the region *reg* each of which moves along a line segment of length $D_{ch}$. We calculate, basing on geometrical analysis, the probability of intersection *Pr*. The probability of intersection is the proportion (on average) between the number of robots $R_j$ such that the chunk of $R_j$ can *possibly* intersect with the chunk of $R_i$, (line segment $A_iB_i$) and the number of robots in the region *reg*. The intersection between the line segment $A_iB_i$ and the other line segments is a function of three variables $(x, y, \theta)$ where: $(x, y)$ are the coordinates of the start point of a line segment, $(x, y) \in reg$ and $\theta$ is the slope of the line segment. The probability of intersection *Pr* equals to the proportion between the volume generated by the tuplet $(x, y, \theta)$ (such that the line segment whose start point is $(x, y)$ and of




(a) The region (*reg*) for the line segment $A_iB_i$.




(b) Intersection probability.

Figure 6.6: Segments intersection.

slope $\theta$ intersects with $A_iB_i$) and the global volume generated by $(x, y, \theta)$ (such that (x, y) scans the region *reg* and $\theta$ rotates from 0 to $2\pi$). The probability $Pr$ is given by the following relation:

$$Pr = \frac{\{(x, y, \theta) \mid Segment(x, y, \theta) \cap A_iB_i \neq \emptyset\}}{\{(x, y, \theta) \mid (x, y) \in reg\}} \tag{6.1}$$

The region *reg* is symmetric with respect to the symmetry axis that is collinear with $A_iB_i$ (Figure. 6.6(b)), so we study the intersection from one side of the symmetry axis. A line segment of slope $\theta$, of length $D_{ch}$ and intersects with $A_iB_i$ must be included in the parallelogram (rhombus) formed by the line segment $A_iB_i$ and the two parallel line segments $A_iC_i$ and $B_iD_i$ of length $D_{ch}$ and of slope $\theta$. The parallelogram is illustrated in Figure. 6.6(b). The area of this parallelogram (rhombus) is: $S_{rho}(\theta) = D_{ch}^2 \sin\theta$, and the area of the region *reg* is $S_{reg} = D_{ch}^2(\pi + 2)$. Consequently, the probability of intersection $Pr$ is:

$$Pr = \frac{2\int_0^{\pi} S_{rho}(\theta)d\theta}{2\pi S_{reg}} = \frac{2}{\pi(\pi + 2)} \quad (\approx 12\%) \tag{6.2}$$

### 6.5.2 Time needed to reserve and move along a chunk

The average physical speed of a robot is denoted by: $V_{mot}$. We calculate the average time required for a robot $R_i$ to reserve and move along a chunk of length $D_{ch}$ with a physical speed $V_{mot}$.

**Number of robots to wait on.** A robot $R_i$ must wait on robots that are present in the region *reg* where collisions may occur. Thus, on average $R_i$ must wait on half the robots existing in the region *reg*, each of which waits also on other robots and so on. Consequently the total number

of robots $n_{avg}$ that $R_i$ must wait on to reserve a chunk is:

$$n_{avg} = \sum_{k=1}^{\infty} (\frac{Pr \cdot n_{reg}}{2})^k \tag{6.3}$$

where $Pr$ is the probability of collision (Subsection. 6.5.1) and $n_{reg}$ is the number of robots in the region *reg*.

In order to minimize the time needed to reserve a chunk, the total number of robots $n_{avg}$ that $R_i$ waits on must be minimal, (we can reduce the value of the reservation range). Thus, the geometric series must be convergent. So, $\frac{Pr \cdot n_{reg}}{2} < 1$, which implies that $n_{reg} < \pi(\pi + 2)$. The number of robots in the region *reg* must be at most $\pi(\pi + 2) \approx 16$ robots, in order to minimize the time to reserve a chunk.

The total number of robots $n_{avg}$ that $R_i$ waits on to reserve a chunk is:

$$n_{avg} = \frac{1}{1 - \frac{n_{reg}}{\pi(\pi+2)}} - 1, \qquad n_{reg} < \pi(\pi + 2) \tag{6.4}$$

**Communication delays.** The average communication delays in the system is denoted: $T_{com}$. When all the robots are active running the protocol (robots do not sleep), then the time needed to reserve and move along a chunk denoted $T_{ch}$ is computed as the sum of the time needed by each of the following steps:

1. The delay $T_{nd}$ of the primitive *NDiscover*.

2. The time elapses until $R_i$ builds the *wait-for* graph, (the local computation time is neglected). $R_i$ needs $2T_{com}$ time units to multicast $Zone_i$ to the neighbors and to receive their requested zones. After that $R_i$ needs $2(n_{avg} - 1)T_{com}$ time units on average, to determine the *dependency* set. (defined in Subsection. 6.2.1). Therefore, the time needed to build the *wait-for* graph is: $2n_{avg}T_{com}$.

3. The time to receive the release messages from $n_{avg}$ robots each of which has owned its zone for $\frac{D_{ch}}{V_{mot}}$ time units is: $n_{avg}(T_{com} + \frac{D_{ch}}{V_{mot}})$.

4. The time needed by $R_i$ to move along a chunk is: $\frac{D_{ch}}{V_{mot}}$.

Therefore, the time needed to reserve and move along a chunk $T_{ch}$ is:

$$T_{ch} = T_{nd} + 2n_{avg}T_{com} + n_{avg}(T_{com} + \frac{D_{ch}}{V_{mot}}) + \frac{D_{ch}}{V_{mot}} \tag{6.5}$$

$$T_{ch} = T_{nd} + n_{avg}(3T_{com} + \frac{D_{ch}}{V_{mot}}) + \frac{D_{ch}}{V_{mot}} \tag{6.6}$$

$$T_{ch} = T_{nd} + (\frac{1}{1 - \frac{n_{reg}}{\pi(\pi+2)}} - 1)(3T_{com} + \frac{D_{ch}}{V_{mot}}) + \frac{D_{ch}}{V_{mot}}, \qquad n_{reg} < \pi(\pi + 2) \tag{6.7}$$

The optimal time $T_{ch}$ for a robot $R_i$ is when it is alone, so there are no robots in the region *reg*. In this case, the time $T_{ch}$ is: $T_{ch}(alone) = T_{nd} + \frac{D_{ch}}{V_{mot}}$.

### 6.5.3 Optimal reservation range

In this subsection, we compute the average effective speed $V$ of a robot $R_i$ as a function of the reservation range and the density of robots, then we determine an optimal value of the reservation range that maximizes the effective speed of $R_i$ for a given value of the density of robots. In our protocol the reservation range is a constant parameter given by the system. The density of robots in the system, denoted by: $s$. ($s = \frac{n_{reg}}{S_{reg}}$). A robot $R_i$ makes on average $\frac{D_{trip}}{D_{ch}}$ steps to move along a path of length $D_{trip}$. The number of robots that $R_i$ has to wait on to reserve a chunk depends on the value of the reservation range and the density of robots. The time to progress a distance $D_{trip}$ is:

$$T_{trip} = \frac{D_{trip}}{D_{ch}}[T_{nd} + (\frac{1}{1 - \frac{n_{reg}}{\pi(\pi+2)}} - 1)(3T_{com} + \frac{D_{ch}}{V_{mot}}) + \frac{D_{ch}}{V_{mot}}], \qquad n_{reg} < \pi(\pi + 2) \tag{6.8}$$

The speed $V$ is $\frac{D_{trip}}{T_{trip}}$, and $n_{reg} = s(\pi + 2)D_{ch}^2$. Thus, the average effective speed $V$ is:

$$V = \frac{-sD_{ch}^3 + \pi D_{ch}}{(3T_{com} - T_{nd})sD_{ch}^2 + \frac{\pi}{V_{mot}}D_{ch} + \pi T_{nd}}, \qquad D_{ch} < \frac{\sqrt{\pi}}{\sqrt{s}} \tag{6.9}$$

The previous relation shows that the effective speed is a function of the reservation range and the density of robots, also the effective speed depends on some system-based fixed parameters such as the communication delays and the physical speed of robots. Figure. 6.7 presents the relationship between the speed and the reservation range for different densities. The values of density start from zero ($R_i$ is alone) until $3[robots/m^2]$.

**Numerical values.** The values of the fixed system parameters are: $T_{com} = 10[ms]$, $T_{nd} = 1[s]$, the physical speed $V_{mot} = 1[m/s]$.

**Speed optimization for a given density.** The first derivative of the function effective speed with respect to the reservation range is:

$$\frac{dV}{dD_{ch}} = \frac{(T_{nd} - 3T_{com})s^2D_{ch}^4 - \frac{2\pi}{V_{mot}}sD_{ch}^3 - \pi(3T_{com} + 2T_{nd})sD_{ch}^2 + \pi^2T_{nd}}{[(3T_{com} - T_{nd})sD_{ch}^2 + \frac{\pi}{V_{mot}}D_{ch} + \pi T_{nd}]^2}, \qquad D_{ch} < \frac{\sqrt{\pi}}{\sqrt{s}} \tag{6.10}$$

The denominator of Equation 6.10 is always positive since $D_{ch} < \frac{\sqrt{\pi}}{\sqrt{s}}$, the speed is maximal when the numerator of Equation 6.10 becomes zero. Figure. 6.7 shows the optimal reservation range for a given density, the value of the optimal reservation range maximizes the effective speed of a robot. The curve that corresponds to the density zero (when robot is alone), in Figure. 6.7, shows that the effective speed always increases as the reservation range increases, until the effective speed $V$ approaches to the physical speed $V_{mot}$ when the value of the reservation range becomes very large. The curve has a horizontal asymptote at $V = V_{mot} = 1[m/s]$. The effective

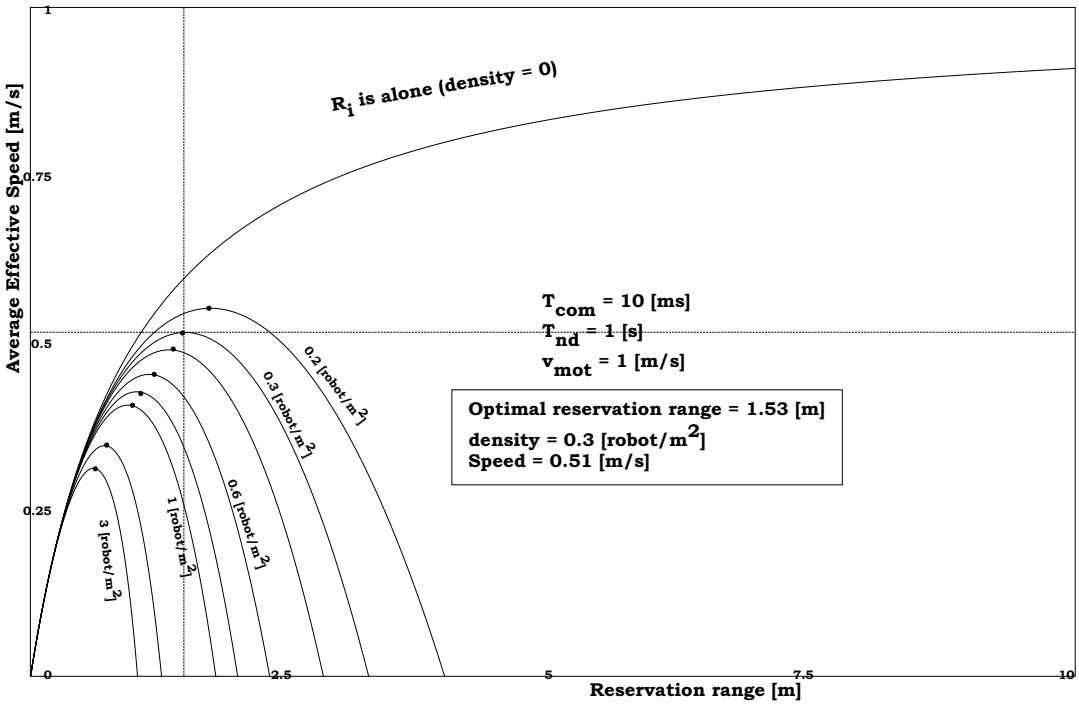



Figure 6.7: Average effective speed vs reservation range.

speed of $R_i$ depends on the reservation range, even in the case when $R_i$ is alone, because it needs to do a certain number of steps to reach a destination, and the number of steps is a function of the reservation range. When the reservation range increases, the number of steps decreases. The relation between the effective speed and the reservation range (when $R_i$ is alone), is the following: $V = \frac{D_{ch}}{\frac{D_{ch}}{V_{mot}} + T_{nd}}$. In each step, $R_i$ needs $T_{nd}$ time units to discover that it is alone. If $D_{ch}$ approaches to infinity, then $V$ approaches to $V_{mot}$.

**Numerical values.** For a density $s = 0.3[robot/m^2]$, the optimal reservation range is $\approx 1.53\ [m]$ which gives a maximal speed $\approx 0.51\ [m/s]$.

### 6.5.4 Speed vs density of robots

In this subsection, we focus on the relation between the effective speed of a robot $R_i$ and the density of robots for a given reservation range. The relationship between the effective speed and the density is presented in Equation 6.9. The derivative of the effective speed with respect to the density of robots is:

$$\frac{dV}{ds} = \frac{-(\frac{\pi}{V_{mot}} D_{ch}^4 + 3\pi T_{com} D_{ch}^3)}{[(3T_{com} - T_{nd})sD_{ch}^2 + \frac{\pi}{V_{mot}} D_{ch} + \pi T_{nd}]^2}, \qquad s < \frac{\pi}{D_{ch}^2} \tag{6.11}$$

Equation 6.11 shows that the effective speed always decreases when the density of robots increases for a given reservation range, as the derivative of the effective speed with respect to

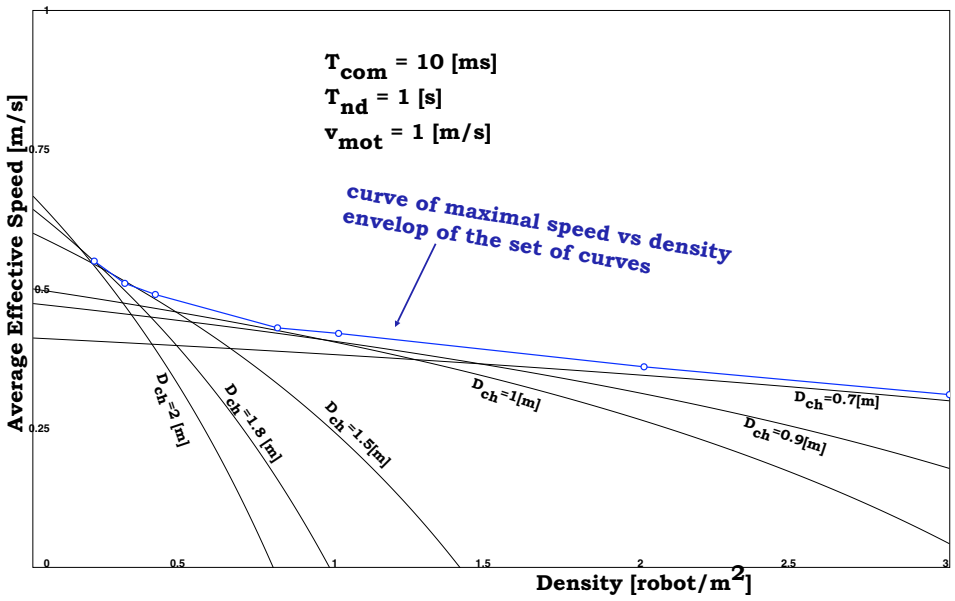



Figure 6.8: Average effective speed vs density of robots.

the density is always negative. Figure. 6.8 presents the relationship between the effective speed and the density for different values of the reservation range, (from 0.7[$m$] to 2[$m$]). The curve that represents the maximal speed as a function of the density, envelops the set of curves in Figure. 6.8. A point that belongs to the envelop corresponds to the optimal reservation range for the given density.

## 6.6 Conclusion

In this chapter, we presented a fail-safe mobility management and achieved a collision prevention platform for asynchronous cooperative mobile robots, in a dynamic group model.

The collision prevention protocol for the dynamic group model, requires neither initial nor complete knowledge of the composition of the group, it relies on a neighborhood discovery primitive which is readily available through most of wireless communication devices. The protocol is based on a locality-preserving distributed path reservation system that takes advantage of the inherent locality of the problem, in order to reduce communication. In the dynamic group model, the design of the collision prevention protocol yields a scalability due to its locality-preserving property. Therefore, the protocol can handle large sized dynamic group of cooperative mobile robots, provided with limited energy resources and limited transmission range.

The dynamic group model is motivated by robotic applications with wide area and large number of robots, where some robots might be out of the transmission range of other robots.

A performance analysis provides insights for a proper dimensioning of system's parameters

in order to maximize the average effective speed of the robots.

The collision prevention platform tolerates the crash of a certain number of robots such that the system of robots keeps in progress towards its final goal, in presence of the crash of a certain number of robots.

The failure of a robot by crash has a *Snowball* effect, because the robots that are waiting for the crashed robot are blocked and the robots that wait for the blocked robots are consequently blocked.

The collision prevention platform for a dynamic group of mobile robots can reach a certain degree of fault-tolerance in a large scale system, due to the locality-preserving property. If a robot crashes then, the local neighbors that are located within one communication hop with respect to the crashed robot at the time of the crash, are blocked waiting on the crashed robot. The two-hop and farther neighbors that do not *conflict* with any of the blocked robots, are not affected at the time of the crash. Therefore, the impact of a crash is limited in space and affects only a part of the system for a period of time, however, the *Snowball* effect takes place with the progress of time.

For the dynamic group model, a non-preemptive fault-tolerant protocol relies on a failure detector of class $\diamond\mathcal{S}$ with the majority of correct robots. If a robot waits for another robot and suspects that the robot has crashed then, the waiting robot cancels its request. In order to cancel a request, a robot executes a Reliable Broadcast of a message, indicating that its request is canceled. The destination of the broadcasted message is the set of robots that conflict with the robot sender. (the zones are intersecting). Reliable Broadcast guarantees that either the message is delivered by all correct robots or none.

# Chapter 7

# Fault-tolerant group membership protocols using physical robot messengers

In this Chapter, we consider a distributed system that consists of a group of teams of worker robots that rely on physical robot messengers for the communication between the teams. Unlike traditional distributed systems, there is a finite amount of messengers in the system, and thus a team can send messages to other teams only when some messenger robot is available locally. It follows that a careful management of the messengers is necessary to avoid the starvation of some teams.

Concretely, this Chapter proposes algorithms to provide group membership and view synchrony among robot teams. We look at the problem in the face of failures, in particular when a certain number of messenger robots can possibly crash.

Consider a robotic mining system composed of mobile robots (worker robots) that cooperate in order to excavate minerals. As there are several excavation sites in the mine, the worker robots are organized into teams, one team working at each site. Teams must coordinate their actions but communication between teams is made difficult by the geography of the mine. For instance, teams are unable to communicate using radio transmission (e.g., [2]) because of propagation problems. Using sound waves for transmission is hard in the presence of echos, and even potentially hazardous in unstable environments. Infrared transmission (e.g., [18]) is impossible without a direct line-of-sight. Wired communication requires a costly infrastructure that may be difficult to deploy in such environment. Thus, to overcome these problems, communication between the teams relies on messenger robots that physically carry messages from one team to the next.

The distributed system modeled in this Chapter consists of robot teams that communicate by exchanging messages carried by physical messengers. Each team has a pool of robot messengers that it can use to carry messages to other teams. When a team has no messenger left in its pool, it is unable to send messages. Conversely, when a messenger coming from another team delivers a message, the number of messengers available in the pool increases by one. Initially,

each team has a given number of messenger in its pool but, in the models we consider, a subset of the messengers can possibly crash during the execution of the system.

This model differs from conventional distributed systems in several aspects. For instance, since a team is unable to communicate when no messenger remain in its pool, a distributed algorithm must adequately balance the use of messengers to avoid deadlock situations wherein the system waits for messages from a team that has become unable to communicate.

We look at two important building blocks of group communication, namely group membership and view synchrony [8]. *Group membership* provides a mechanism to allow teams to dynamically join or leave the system during an execution. Membership changes are done consistently by all teams, through the installment of group membership views. A view consists of a view sequence number and the list of teams that belong to the group. As all teams must agree on every view installed, they also agree on the composition of the group for any given view. *View synchrony* relates to broadcast communication among the group members. In short, it ensures that a message broadcasted within a view is received by all teams before they install the next view, thus providing a form of "all-or-nothing" semantics in the face of failures.

**Related work** There exist many definitions of group membership and view synchrony, discussed extensively by Chockler et al. [8]. Correspondingly, there are many systems supporting some form of group membership and view synchrony in conventional distributed systems, such as Isis [3], Transis [12], Totem [24], Moshe [19], and many others (e.g., [13, 1]). In mobile robotics, Schemmer et al. [27] present two membership protocols for robots communicating through wireless network protocols such as 802.11. However, none of these group membership protocols consider a system where communication uses physical messengers, and they are hence not adapted to such environments.

In contrast, mobile agent systems consider mobile entities that can carry information. There is however a very fundamental difference between mobile agents and robot messengers. While the former are software entities that can be easily replicated or regenerated programmatically, the latter are physical and cannot be easily recovered after a failure. Thus, protocols developed for mobile agents cannot easily be adapted to our system model.

This Chapter identifies the importance of a distributed systems relying on physical messengers, and to provide a group membership and view synchrony protocol for such systems. This Chapter presents in fact two protocols, where the first one tolerates the failure of a bounded number of messengers, while the second one additionally tolerates the failure of entire teams. We present the protocols, provide arguments to support their correctness, and discuss their complexity in terms of time and energy consumption.

This Chapter is organized as follows. Section 7.1 presents the system model and basic def-

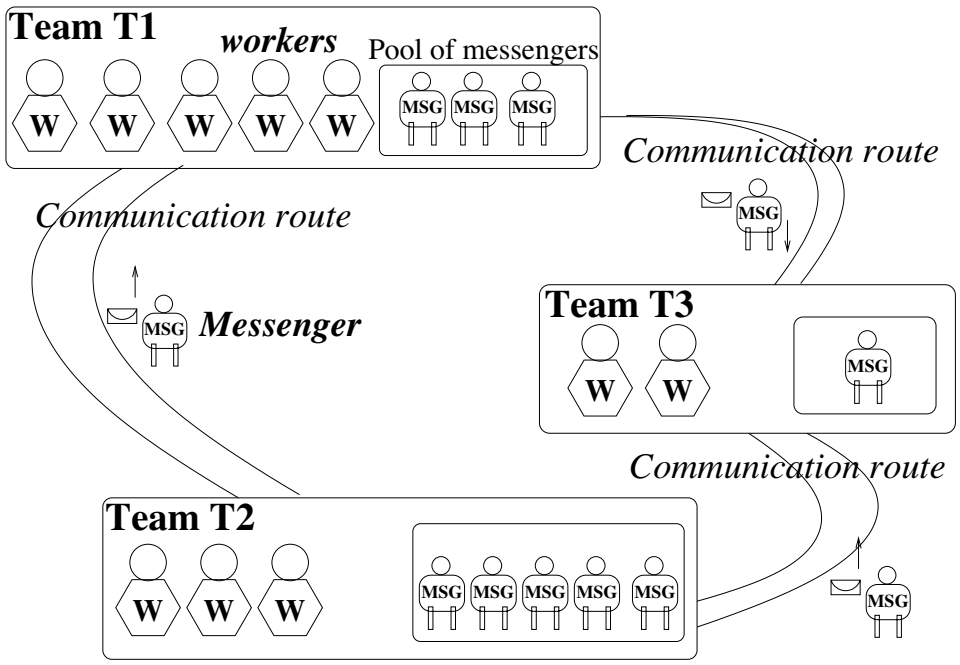



Figure 7.1: System model.

initions. Section 7.2 describes two failure models; one in which messengers can fail, and a second one in which teams can also fail. Section 7.3 presents and discusses two group membership algorithms, one for each failure model. Finally, Section 7.4 concludes this Chapter.

## 7.1 System model & definitions

### 7.1.1 System model

We model a distributed system as a set of teams of autonomous mobile robots $S = \{T_1, \ldots, T_n\}$ and a set of $m$ messengers, where $m > 1$. Every team has an identifier, a set of robots named "workers" responsible for executing the required tasks, and a pool of robot messengers.

Communication between teams is done by sending a messenger from the pool to convey messages. Messengers travel from a team to another by following a "communication route." We assume that a direct route exists between any pair of teams, although this assumption is in fact not restrictive since two routes can physically share the same path. When a messenger delivers its last message at a team $T_i$, the messenger joins the pool of team $T_i$. The capacity of a messenger is assumed to be large enough to accommodate any finite number of messages.

The system is purely asynchronous in the sense that timing assumptions are made neither on processing speed nor on the time it takes for a messenger to carry a message from a team to another.

Figure 7.1.

### 7.1.2 Energy complexity

In addition to the complexity metrics used in traditional distributed systems, we consider an additional complexity metric that we call *energy complexity*.

Energy complexity counts the total number of *"hops"* traveled by messengers during a single execution of the algorithm. A hop is the journey of one messenger from one team to another. Roughly speaking, the energy complexity of an algorithm corresponds to the energy spent by the algorithm during a single run.

### 7.1.3 Group membership & view synchrony

The membership service maintains a list of currently active teams, in failure-prone distributed systems, and delivers this information to the application whenever its composition changes. The reliable multicast services deliver messages to the current view members. For more information on the subject, we refer to the survey of Chockler et al. [8]. A group membership can also be combined with failure detection, and then it can be seen as a high-level failure detection mechanism that provides consistent information about suspicions and failures [30, 17]. In short, a group membership keeps a track of what teams belong to the distributed computation and what team does not.

In our model, a group membership service provides a list of non-crashed teams that currently belong to the system, and satisfies three properties [8]: validity, agreement and termination. Validity is explained as follows: let $v_i$ and $v_{i+1}$ be two consecutive views, if a team $T_i \in v_i \setminus v_{i+1}$ then some team has executed *leave($T_i$)* and if a team $T_i \in v_{i+1} \setminus v_i$ then some team has executed *join($T_i$)*. The agreement property ensures that the same view would be installed by all the teams of the group (agreement on the view) since agreement on uniquely identified views is necessary for synchronizing communications. Termination means that if team $T_i$ executes join($T_q$), then unless $T_i$ crashes, eventually a view $v'$ is installed such that either $T_q \in v'$ or $T_p \notin v'$. We present the following notations used in this Chapter:

- $|T_i|$ is the number of messengers exist in the pool of team $T_i$.
- initiator is the team which proposes (join) or a (leave) operation, and consequently initiates a procedure of creating a new view.

- logical ring is a logical circular list of teams identifiers.
- $v_{ini}$ is the initial view of the system.
- $v_{act}$ is the current view of the system.
- $v_{fin}$ is the resulting view of the system.

## 7.2 Failure Models

We consider that a messenger and a team can fail by crashing and that crashes are permanent. We look at two slightly different failure models. In Model A, we consider the crash of messengers only, but assume that all teams are correct. We then look at a more general model (Model B), in which both teams and messengers can possibly crash.

### 7.2.1 Model A: Messengers failures

In Model A, we consider the failure of messengers only, so a robot messenger may crash, for instance while it travels from one team to another. However, it is assumed that team do not crash.

We assume that the total number of faulty messengers is bounded, and denote this upper bound by $\overline{M}$.

In this model, we have the following properties:

- *property A1*: A messenger can fail by crashing, and when it crashes, the crash is permanent.
- *property A2*: Teams never fail.
- *property A3*: There is at least one correct messenger in the system, so ($\overline{M} < m$).

### 7.2.2 Model B: Teams/messengers failures

In Model B, we extend Model A by also considering that teams may fail. We assume that the number of faulty teams and faulty messengers is bounded, and we denote the maximal number of faulty teams, respectively messengers, by $\overline{T}$, respectively $\overline{M}$.

In this model, we have the following properties:

- *Property B1*: A whole team(s) may fail by crashing, and when a crash occurs, it is permanent.

- *Property B2*: Correct messengers never crash. Nevertheless, a correct messenger can be rendered nonfunctional, as covered by Property B3.
- *Property B3*: When a team crashes, none of the messengers that were in its pool at the instant of the crash can move. The rationale is that, without a proper source of energy provided by the pool, the messengers are unable to travel to other teams. This affects all messengers in the pool, regardless whether they are faulty or correct. However, messengers that were in transit at the moment of the crash are not affected.
- *Property B4*: There is at least one correct team in the system, and at least one correct robot messenger in the system, we can express this condition as follows: $\overline{T} < n$ and $\overline{M} < m$.
- *Property B5*: Any set composed of $\overline{T}$ teams, should contain at most $m - \overline{M} - 1$ robot messengers in total, at any instant.

The motivation behind Property B5 is to guarantee the existence of at least one correct robot messenger in the system, even if $\overline{T}$ simultaneous crashes occur.

## 7.3 Group Membership and View Synchrony algorithms

In this section, we study the problem of group membership and view synchrony in our system model, considering the two precedent failure models.

For each failure model, we give a brief explanation and illustrate our algorithm by an example, then we give arguments showing its correctness, and finally we evaluate the energy and time required to run the algorithm.

We represent the system as a logical ring of nodes sorted by increasing order of teams identifiers, each node in the list represents a team of robots, the initial view contains all the teams in the system.

### 7.3.1 Group membership & messengers failures (Model A)

We study the group membership and view synchrony in our system model, in presence of messengers failures.

#### Description of the algorithm (Model A)

- *Condition*: The team initiator has initially at least $(\overline{M} + 1)$ messengers in its pool.

In Model A, the team initiator executes the propose and commit rounds by sending a set of messengers which has at least one correct. We illustrate the algorithm by the following simple example:

Consider a system composed of three teams, we construct a logical ring of nodes $\{T_1, T_2, T_3\}$. The initial view is: $\{T_1, T_2, T_3\}$. We suppose that the team ($T_2$) starts to propose a new view (team initiator), and it invokes *join($T_p$)* operation, the team $T_3$ invokes a *leave($T_3$)* operation, and $T_1$ does not execute any operation. The team $T_2$ starts the propose round by sending a set of $(\overline{M} + 1)$ messengers to $T_3$, such that each messenger carries the same message which proposes the view: $v^i_{T_2} = \{T_1, T_2, T_3, T_p\}$.

When the team $T_3$ receives this set of messengers (or at least one), it behaves as follows:

1. unifies all the identical messages received from $T_2$.
2. generates its own message: $T_3.leave(T_3)$.
3. merges $msg_{T_2}$ and $msg_{T_3}$, then proposes the view $v^i_{T_3} = \{T_1, T_2, T_p\}$.
4. sends the set of messengers that it has received to $T_1$, with the new view $v^i_{T_3}$.

When $T_1$ receives the set of messengers from $T_3$, it does not change the view, acknowledges the current proposed view, and sends the messengers to $T_2$, which terminates the propose round and starts the commit round when it receives at least one messenger from $T_1$. $T_2$ starts the commit round by sending the same set of messengers with the message $commit\{T_1, T_2, T_p\}$ to $T_3$ which acknowledges the current view $v^i$ and sends the set to $T_1$.

The algorithm terminates when $T_2$ receives back at least one messenger of this set carrying the commit message that it has sent, and the group membership algorithm is successfully terminated, such that the team $T_p$ has joined the group and $T_3$ has left it, and the new view is $v^i = \{T_1, T_2, T_p\}$.

**Correctness arguments (Model A)**

The condition $|initiator| \geq (\overline{M} + 1)$ guarantees that the team initiator has at least **one correct** messenger, we show that this condition ensures the correct termination of the algorithm.

In the Model A, we need to send $(\overline{M} + 1)$ messengers from the initiator to the next team, in order to guarantee the correct reception of messages by the next team, [1] supposing that all the teams are correct. (assumptions of this failure model)

The cardinality of this set remains larger than 1 and smaller or equal to $(\overline{M} + 1)$ because some messengers may crash before reaching their destinations, and this set of messengers is responsible for all the communications between the teams until return back to the initiator. (propose and commit rounds).

[1] The set of messengers moving between teams may become smaller after each step of the algorithm, but this set is never empty.

The algorithm guarantees that the same new view is acknowledged by all the teams in the system, since there is no team crash in this model.

The properties: Validity, Agreement, and Termination are discussed exactly as in the failure-free model presented in our research report [33].

**Behavior Evaluation (Model A)**

The initiator sends a set of $(\overline{M}+1)$ messengers, and this same set performs $2n$ hops between the teams in order to define a new view, so the energy consumed is in the order of $O(2(\overline{M}+1)n)$. But the time required is $O(2n)$ because the robots move simultaneously.

### 7.3.2 Group membership & teams and messengers failures (Model B)

We study the algorithm of group membership in presence of both teams and messengers failures.

**Description of the algorithm (Model B)**

- *Condition 1*: The team initiator is a correct team.
- *Condition 2*: The team initiator has initially at least $(\overline{M}+1)$ messengers in its pool.

We illustrate the group membership algorithm by the following simple example:

Consider a system composed of four teams, we construct a logical ring of nodes $\{T_1, T_2, T_3, T_4\}$. The initial view is $v_{ini} = \{T_1, T_2, T_3, T_4\}$. We suppose that the team $T_2$ starts to propose a new view (team initiator), and it invokes the *join($T_p$)* operation. For simplicity, we suppose that other teams do not execute any operation, and the teams $T_1$ and $T_3$ are faulty.

**propose round** The team $T_2$ starts the propose round by sending a set composed of $(\overline{M}+1)$ messengers to $T_3$, such that each messenger in the set carries the same message which proposes the view $v^i_{T_2} = \{T_1, T_2, T_3, T_4, T_p\}$. When this set of messengers arrives to the site of $T_3$, it performs a crash detection protocol based on hand-shaking with all the workers of $T_3$. There are two cases:

- $T_3$ *has crashed*: the set of messengers returns back to $T_2$ indicating the crash of $T_3$, then the initiator changes the current view by removing $T_3$ from the group (forced leave) and sends this set of messengers to $T_4$, carrying the newly proposed view $v^i_{T_2} = \{T_1, T_2, T_4, T_p\}$.
- $T_3$ *is alive*: it unifies the identical messages, and sends the set of messengers to $T_4$, as in Model A.

The propose round terminates when the team initiator receives back its set of messengers (or part of it).

**commit round** $T_2$ starts the commit round by sending the set of messengers with the message $commit\{T_1, T_2, T_4, T_p\}$ to $T_4$, which acknowledges the current view $v^i$ and sends the set to $T_1$. When the messengers arrive to the site of $T_1$ they can face two cases:

- $T_1$ *has crashed*: the set of messengers returns back to $T_2$ indicating the crash of $T_1$, then the initiator changes the current view by removing $T_1$ from the group (forced leave) and restarts the commit round by sending this set of messengers to $T_4$ again, provided with the current commit view $v^i_{T_2} = \{T_2, T_4, T_p\}$. Then $T_4$ acknowledges the commit view and sends the messengers to $T_2$.

- $T_1$ *is alive*: it unifies the identical messages, and sends the set of messengers to $T_2$, as in Model A.

The algorithm terminates when $T_2$ receives back at least one messenger belongs to the set it has sent, provided with the commit message, and the group membership algorithm is successfully terminated, such that the team $T_p$ has joined the group and $(T_1, T_3)$ have left it because of their crashes, and the new view is $v^i = \{T_2, T_4, T_p\}$.

**Correctness arguments (Model B)**

In this model, we have team failures in addition to messenger failures, so we need extra specifications concerning a team correctness.

We show that the two previous conditions guarantee that the algorithm terminates correctly. In our model a messenger can perform at most two hops, so when a messenger moves to a crashed team, the next hop should be to a correct one, else the messenger would be idle. The messenger returns to the team initiator after detecting a crashed team, and the initiator is a correct team according to (Condition 1), while (Condition 2) guarantees that the set of messengers sent by the initiator, has at least one correct messenger. This set performs all the hops between the teams as we discussed for Model A.

In this model, we provide the system with a perfect failure detector, because the detection of a crashed team is carried out by a local hand-shaking mechanism, between at least one correct messenger and all the workers of the team. After detecting a crashed team by a messenger, this messenger moves to the team initiator (correct team), and proclaims the crashed team, consequently, the crash is detected correctly and deterministically.

The commit round, permits to provide each non-crashed-team with the most recent view, because the initiator restarts the commit round whenever it detects a crashed team, so the commit round terminates correctly by delivering the same view $v_{fin}$ to all non-crashed teams.

The properties: Validity, Agreement, and Termination are discussed exactly as in the failure-free model presented in our research report [33].

**Behavior Evaluation (Model B)**

The set of $(\overline{M}+1)$ messengers may perform additional hops because of teams failures, so this set needs to go backward to the initiator whenever it detects a crashed team. (additional $\overline{T}$ hops in the propose round, and $n \cdot \overline{T}$ hops during the commit round). The energy consumed by messengers is calculated as follows:

Propose round(worst case):$(n+\overline{T})(\overline{M}+1)$.

Commit round(worst case):$(n \cdot \overline{T})(\overline{M}+1)$.

The total energy consumption is $(\overline{M}+1)(\overline{T}+1) \cdot n + \overline{T}(\overline{M}+1)$.

The behavior evaluation in terms of energy can be written as : $O(\alpha \cdot n + \beta)$.

The behavior in terms of time is $(\overline{T}+1) \cdot n + \overline{T}$, also it can be expressed as: $O(\lambda \cdot n + \mu)$.

**Discussion** The required energy and time to run the algorithm increase when fault-tolerance requirements become harder. In the failure-free model [33] the algorithm requires energy and time proportional to $(2n)$, where $n$ is the size of the system (group of teams). In Model A the energy becomes more significant, it is $\overline{M}$ times greater, which is justified by the cost required to tolerate the messengers failures, but the execution time is equivalent to that in failure-free model [33]. When the system is prone to teams failures in addition to messengers failures in Model B, the required energy is $(\overline{M} \cdot \overline{T})$ times greater than that in the failure-free model [33], while the execution time is only $\overline{T}$ times greater.

## 7.4 Conclusion

We have introduced a distributed asynchronous system model composed of a group of teams of cooperative mobile robots. The teams in our model communicate by physical robot messengers. We have presented a group membership algorithm, discussed its correctness, and evaluated its behavior in terms of energy and time, in two possible failure models: one in which only messengers fail, and one in which entire teams can also fail.

We have shown the conditions that should be satisfied to solve the problem of group membership in our system model in each different failure model. This technique of communications between teams of robots permits to implement a perfect failure detector since the detection of a crashed team takes place locally on its site. This property permits to solve many agreement problems in asynchronous distributed systems composed of a group of teams of robots.

Furthermore, in this model faulty robot messengers can be mapped to lossy channels in classical distributed systems. We guaranteed reliable communications by using one set of messengers that has at least a correct messenger, this set circulates the messages and supports all the communications between the teams of the system.

**Algorithm 20** Algorithm(A) Group membership: messengers failures (code for Team $T_i$)

1: Initialisation:
2: $S \leftarrow \{T_1, T_2, \ldots, T_n\}$
3: $v_{ini} \leftarrow \{T_i, i \in [1..n]\}$
4: $v_{act} \leftarrow v_{ini}$
5: $old\_view \leftarrow v_{ini}$
6: $msg \leftarrow \emptyset$
7: operation = {join(new_team), leave(team)}

8: **if** (operation = join (new_team)) **then**
9: $T_i$.propose ($v_{act} \leftarrow v_{act} \bigcup \{new\_team\}$)
10: **end if**
11: **if** (operation = leave(team)) **then**
12: $T_i$.propose ($v_{act} \leftarrow v_{act} \setminus \{team\}$)
13: **end if**
14: **if** ($T_i = initiator$) and ($|initiator| \geq (\overline{M} + 1)$) **then**
15: $msg.initiator \leftarrow initiator(ID)$
16: $msg \leftarrow msg \bigcup \{v_{act}\}$
17: send set of $(\overline{M} + 1)$ messengers provided with (msg) to next($T_i$)
18: **wait until** reception of the messengers sent
19: **when** reception of messengers sent
20: begin-commit-round()
21: **end when**
22: **end if**
23: **if** ($T_i \neq initiator$) **then**
24: unify all the identical propose messages received from previous($T_i$) into one message (msg)
25: $msg \leftarrow msg \bigcup \{v_{act}\}$
26: send **the set of messengers received** from previous($T_i$) provided with (msg) to next($T_i$)
27: **end if**
28: **procedure** begin-commit-round
29: **if** ($T_i = initiator$) **then**
30: $v_{fin} \leftarrow v_{act}$
31: $msg \leftarrow commit(v_{fin})$
32: send **the set of messengers** received from previous($T_i$) provided with (msg) to next($T_i$)
33: **wait until** reception of the messengers sent
34: **when** reception of messengers sent
35: terminate-commit-round()
36: **end when**
37: **else**
38: unify all the identical commit messages received from previous($T_i$) into one message (msg)
39: send **the set of messengers** received from previous($T_i$) provided with (msg) to next($T_i$)
40: **end if**
41: $new\_view \leftarrow v_{fin}$
42: **end**begin-commit-round

**Algorithm 21** Algorithm(B) Group membership: Teams and messengers failures (code for Team $T_i$)

1: Initialisation:
2: $S \leftarrow \{T_1, T_2, \ldots, T_n\}$
3: $v_{ini} \leftarrow \{T_i, i \in [1..n]\}$
4: $v_{act} \leftarrow v_{ini}$
5: $old\_view \leftarrow v_{ini}$
6: $msg \leftarrow \emptyset$
7: operation = {join(new_team), leave(team)}

8: **if** (operation = join (new_team)) **then**
9: $T_i$.propose ($v_{act} \leftarrow v_{act} \bigcup \{new\_team\}$)
10: **end if**
11: **if** (operation = leave(team)) **then**
12: $T_i$.propose ($v_{act} \leftarrow v_{act} \setminus \{team\}$)
13: **end if**
14: **if** ($T_i = initiator$) and ($|initiator| \geq (\overline{M} + 1)$) **then**
15: $msg.initiator \leftarrow initiator(ID)$
16: $msg \leftarrow msg \bigcup \{v_{act}\}$
17: send set of $(\overline{M} + 1)$ messengers provided with (msg) to next($T_i$)

18: **when** reception of messengers carrying the failure detection message
19: $v_{act} \leftarrow v_{act} \setminus \{T_k\}$ $\{Propose\ round\}$
$\{T_k$ *has crashed and the current message is (msg)*$\}$
20: $msg \leftarrow msg \bigcup \{v_{act}\}$
21: **if** ($next(T_k) \neq initiator$) **then**
22: send **the received set of messengers** carrying (msg) to next($T_k$)
23: **else**
24: begin-commit-round()
25: **end if**
26: **end when**

27: wait until reception of messengers sent.
28: **when** reception of messengers carrying a "non failure-detection message"
29: begin-commit-round()
30: **end when**
31: **end if**
32: **if** ($T_i \neq initiator$) **then**
33: unify all the received identical propose messages into one message (msg).
34: $msg \leftarrow msg \bigcup \{v_{act}\}$
35: send **the received set of messengers** carrying (msg) to next($T_i$)
36: **end if**

**Algorithm 22** begin-commit-round (code for Team $T_i$)

```
1:  procedure begin-commit-round()
2:    if (T_i = initiator) then
3:      v_fin ← v_act
4:      msg ← commit(v_fin)
5:      list ← list \ {detected_crashed_teams}   {v_fin is identical to the updated logical ring of teams identifiers,
        so when a team receives the commit message, it discovers the next team to send.}
6:      if (the number of detected crashed teams < n-1) then
7:        send the received set of messengers carrying (msg) to next(initiator) in the new logical ring.
8:      else
9:        terminate-commit-round()   {the number of detected crashed teams = n-1, i.e. the initiator is the only
          correct team in the system}
10:     end if

11:     when reception of messengers carrying the failure detection message
12:       v_act ← v_act \ {T_k}                                                    {Commit round}
                                                  {T_k has crashed and the current message is (msg)}
13:       msg ← msg ⋃ {v_act}
14:       begin-commit-round                                                       {restart from the beginning}
15:     end when

16:     wait until reception of messengers sent
17:     when reception of messengers carrying a "non failure-detection message"
18:       terminate-commit-round()
19:     end when
20:   else
21:     unify all the received identical commit messages into one message (msg)
22:     send the received set of messengers carrying (msg) to next(T_i) in the new logical ring.
23:   end if
24:   new_view ← v_fin
25: end begin-commit-round
```

**Algorithm 23** Messenger (code for messenger)

```
1: if detect_crash(messenger, T_k) then
2:   move to the team initiator.   {if a messenger detects that a team has crashed then, the messenger returns to
     the team initiator}
3: end if
```

The model presented in this Chapter was motivated by a mining application in which teams of worker robots cooperate to excavate minerals. The model is however by no means limited to this application. In fact, a similar model applies to robot applications in underwater environment, deep space exploration, nano-scale robotic devices, or more generally in environments where there are no established communication infrastructures.

# Chapter 8

# Conclusion

In this dissertation, we presented a fail-safe mobility management and achieved a collision prevention platform for asynchronous cooperative mobile robots.

This platform consists of time-free collision prevention protocols, that ensure the safety property of the system and guarantee that no collision can occur between mobile robots, whatever the temporal guarantees offered by the communications system.

We have presented collision prevention protocols for a closed group of mobile robots and for a dynamic group. The closed group model of mobile robots is motivated by robotic applications with low number of robots in a limited space, in which mobile robots are always within the transmission range of each other, and each robot knows the total composition of the group. However, in wide area mobile robotic applications with large number of robots, the communication connectivity between mobile robots is not guaranteed. In such applications, mobile robots can rely on the collision prevention platform for a dynamic group model.

The collision prevention protocol for the dynamic group model, requires neither initial nor complete knowledge of the composition of the group, it relies on a neighborhood discovery primitive which is readily available through most of wireless communication devices. The protocol is based on a locality-preserving distributed path reservation system that takes advantage of the inherent locality of the problem, in order to reduce communication. Therefore, the protocol for a dynamic group model is scalable by design, while the scalability of the protocol for a closed group model is low. The collision prevention protocol for a closed group invokes all the robots in the system.

The collision prevention protocol for a closed group is robust with respect to time, while the vulnerability to time resides only in the neighborhood discovery primitive used by the collision prevention protocol for a dynamic group of robots. On the other hand, the closed group model requires a total communication connectivity between mobile robots and also each robot must know the total composition of the group.

In the dynamic group model, the design of the collision prevention protocol yields a scal-

ability due to its locality-preserving property. Therefore, the protocol can handle large sized dynamic group of cooperative mobile robots, provided with limited energy resources and limited transmission range.

A performance analysis provides insights for a proper dimensioning of system's parameters in order to maximize the average effective speed of the robots.

The collision prevention platform tolerates the crash of a certain number of robots such that the system of robots keeps in progress towards its final goal, in presence of the crash of a certain number of robots.

The failure of a robot by crash has a *Snowball* effect, because the robots that are waiting for the crashed robot are blocked and the robots that wait for the blocked robots are consequently blocked.

The collision prevention platform for a dynamic group of mobile robots can reach a certain degree of fault-tolerance in a large scale system, due to the locality-preserving property. If a robot crashes then, the local neighbors that are located within one communication hop with respect to the crashed robot at the time of the crash, are blocked waiting on the crashed robot. The two-hop and farther neighbors that do not *conflict* with any of the blocked robots, are not affected at the time of the crash. Therefore, the impact of a crash is limited in space and affects only a part of the system for a period of time, however, the *Snowball* effect takes place with the progress of time.

Chapter 5 presented two fault-tolerant collision prevention protocols rely on $\diamond\mathcal{S}$ failure detector and tolerate the failures of half of the robots. One of the fault-tolerant collision prevention protocols is preemptive in the sense that a request of a robot can be preempted due to a decision voted by the other robots of the group, if the robot is suspected as a crashed robot. The second protocol is non-preemptive, so a robot $R_i$ cancels its own request when it suspects another robot $R_j$ as a crashed robot, if $R_i$ waits for $R_j$.

Therefore, if a robot $R_i$ suspects that a robot $R_j$ has crashed, and $Zone_i$ intersects with $Zone_j$ then, $R_i$ cancels the request ($R_i$, $Zone_i$) and restarts a request of an alternative zone. Other robots that wait for $R_i$ continue the execution of the protocol. This technique is applied for both group models, the closed group model as well as the dynamic group model.

For the dynamic group model, a non-preemptive fault-tolerant protocol relies on a failure detector of class $\diamond\mathcal{S}$ with the majority of correct robots. If a robot cancels its request then, it executes a Reliable Broadcast of a message indicating that its request is canceled. The destination of the broadcasted message is the set of robots that conflict with the robot sender. (the zones are intersecting). Reliable Broadcast guarantees that either the message is delivered by all correct robots or none.

This dissertation also provides group membership and view synchrony protocols among

robot teams, in a distributed system model composed of a group of teams of worker robots that rely on physical robot messengers for the communication between the teams. The protocols tolerate the crash of a certain number of messengers robots and the crash of a certain number of teams. Unlike traditional distributed systems, there is a finite amount of messengers in the system, and thus a team can send messages to other teams only when some messenger robot is available locally.

Chapter 7 presents a group member a group membership algorithm, discussed its correctness, and evaluated its behavior in terms of energy and time, in two possible failure models: one in which only messengers fail, and one in which entire teams can also fail.

In Chapter 7, we have shown the conditions that should be satisfied to solve the problem of group membership in our system model in each different failure model. This technique of communications between teams of robots permits to implement a perfect failure detector since the detection of a crashed team takes place locally on its site. This property permits to solve many agreement problems in asynchronous distributed systems composed of a group of teams of robots.

Furthermore, in this model faulty robot messengers can be mapped to lossy channels in classical distributed systems. We guaranteed reliable communications by using one set of messengers that has at least a correct messenger, this set circulates the messages and supports all the communications between the teams of the system.

The model presented in Chapter 7, was motivated by a mining application in which teams of worker robots cooperate to excavate minerals. The model is however by no means limited to this application. In fact, a similar model applies to robot applications in underwater environment, deep space exploration, nano-scale robotic devices, or more generally in environments where there are no established communication infrastructures.

# Bibliography


[1] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. Rtcast: Lightweight multicast for real-time process groups. In *Proc. IEEE. Symp. on Real-Time Technology and Applications (RTAS'96)*, pages 250–259, Boston, MA, USA, 1996.

[2] B. Bellur, M. Lewis, and F. Templin. An ad-hoc network for teams of autonomous vehicles. In *Proc. Symp. on Autonomous Intelligent Networks and Systems (AINS-02)*, Los Angeles, USA, May 2002.

[3] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[4] W. Burgard, M. Moors, and F. Schneider. Collaborative exploration of unknown environments with teams of mobile robots. *Springer Verlag*, 2002.

[5] Y. Cao, A. Fukunaga, and A. Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4(1):7–27, 1997.

[6] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM: Journal of the ACM*, 43(4):685–722, 1996.

[7] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM: Journal of the ACM*, 43(2):225–267, 1996.

[8] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv*, 33(4):427–469, December 2001.

[9] X. Défago. Distributed computing on the move: From mobile computing to cooperative robotics and nanorobotics. In *Proc. ACM Int'l Workshop on Principles of Mobile Computing (POMC'01)*, pages 49–55, Newport, RI, USA, August 2001.

[10] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[11] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *J. ACM: Journal of the ACM*, 34(1):77–97, 1987.

[12] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communnication. *Communications of the ACM*, 39(4):64–70, April 1996.

[13] P. Ezhilchelvan and R. de Lemos. A robust group membership algorithm for distributed real-time systems. In *Proc. 11th IEEE. Symp. on Real-Time Systems (RTSS '90)*, pages 173–181, Lake Buena Vista, Florida, USA, December 1990.

[14] M. Fischer, N. Lynch, and M.Paterson. Impossibility of distributed consensus with one faulty process. *JACM: Journal of the ACM*, 32, 1985.

[15] V. Hadzilacos and S. Toueg. *Distributed Systems*. Addison-Wesley, Second Edition, 1993.

[16] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

[17] N. Hayashibara, X. Défago, R. Yared, and T. Katayama. The $\varphi$ accrual failure detector. In *Proc. 23nd IEEE Intl. Symp. on Reliable Distributed Systems (SRDS'04)*, Florianópolis, Brazil, October 2004.

[18] H. Hu, I. Kelly, D. Keating, and D. Vinagre. Coordination of multiple mobile robots via communication. In *Proc. 13th IEEE Intl. Conf. on Mobile Robots (SPIE'98)*, pages 94–103, Boston, USA, November 1998.

[19] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for WANs. *ACM Trans. Comput. Syst.*, 20(3):191–238, August 2002.

[20] P. Martins, P. Sousa, A. Casimiro, and P. Veríssimo. Dependable adaptive real-time applications in wormhole-based systems. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN'04)*, Florence, Italy, June 2004.

[21] P. Martins, P. Sousa, A. Casimiro, and P. Veríssimo. A new programming model for dependable adaptive real-time applications. *IEEE Distributed Systems Online*, 6(5), May 2005.

[22] J. Minguez and L. Montano. Nearness diagram (ND) navigation: Collision avoidance in troublesome scenarios. *IEEE Trans. on Robotics and Automation*, 20(1):45–59, 2004.

[23] L. Montano and J. Asensio. Real-time robot navigation in unstructured environments using a 3D laser rangefinder. In *IEEE/RSJ Conf. on Intelligent Robots and Systems*, pages 526–532, 1997.

[24] L. E. Moser, P. M. Melliar-Smith, D. A. Agrawal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[25] E. Nett and S. Schemmer. Reliable real-time communication in cooperative mobile applications. *IEEE Trans. Computers*, 52(2):166–180, 2003.

[26] E. Nett and S. Schemmer. An architecture to support cooperating mobile embedded systems. In *ACM Intl. Conf. on Computing Frontiers (CF'04)*, pages 40–50, Ischia, Italy, April 2004.

[27] S. Schemmer and E. Nett. Managing dynamic groups of mobile systems. In *Proc. 6th IEEE Intl. Symp. on Autonomous Decentralized Systems (ISADS'03)*, pages 9–16, Pisa, Italy, April 2003.

[28] A. Segall. Distributed network protocols. *IEEE Trans. on Inf. Theory (IT-29)*, pages 23–35, 1983.

[29] R. Simmons. The curvature-velocity method for local obstacle avoidance. In *IEEE/RSJ Conf. on Intelligent Robots and Systems*, pages 3375–3382, 1996.

[30] P. Urbán, I. Shnayderman, and A. Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN'03)*, pages 645–654, San Francisco, CA, USA, June 2003.

[31] P. Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, pages 108–113, 2003.

[32] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Trans. Computers*, 51(8):916–930, 2002.

[33] R. Yared, X. Défago, and T. Katayama. Fault-tolerant group membership protocols using physical robot messengers. Research Report IS-RR-2004-019, Japan Advanced Institute of Science and Technology (JAIST), Hokuriku, Japan, December 2004.

[34] R. Yared, X. Défago, and T. Katayama. Fault-tolerant group membership protocols using physical robot messengers. In *19th IEEE Int'l Conf. on Advanced Information Networking and Applications (AINA)*, pages 921–926, Taipei, Taiwan, March 2005.

# Publications

[1] R. Yared, X. Défago, and T. Katayama.
Fault-tolerant group membership protocols using physical robot messengers.
In *19th IEEE Int'l Conf. on Advanced Information Networking and Applications (AINA'05)*, pages 921–926, Taipei, Taiwan, March 2005.

[2] N. Hayashibara, X. Défago, R. Yared, and T. Katayama.
The $\varphi$ Accrual Failure Detector. In *Proc. 23nd IEEE Intl. Symp. on Reliable Distributed Systems (SRDS'04)*, Florianópolis, Brazil, October 2004.

**Invited papers**

- R. Yared, X. Défago, and T. Katayama.
Distributed collision freedom protocol for a group of autonomous mobile robots. In *JSF workshop (Journées Scientifiques Francophones) JSF 2005*, Tokyo, November 2005.

- R. Yared, X. Défago, and T. Katayama.
Failure detection and group communications for autonomous mobile systems. In *JSF workshop (Journées Scientifiques Francophones) JSF 2004*, Tokyo, November 2004.

**Research reports**

- R. Yared, J. Cartigny, X. Défago, and M. Wiesmann.
Locality-preserving distributed collision prevention protocol for asynchronous cooperative mobile robots. *Research Report IS-RR-2006-003*, JAIST, Japan, February 2006.

- R. Yared, X. Défago, and T. Katayama.
Fault-tolerant group membership protocols using physical robot messengers. *Research Report IS-RR-2004-019*, JAIST, Japan, November 2004.