Fault-Tolerant Dispersion of Mobile Robots

Prabhat Kumar Chand(✉), Manish Kumar, Anisur Rahaman Molla, and Sumathi Sivasubramaniam

Indian Statistical Institute, Kolkata, India
pchand744@gmail.com, manishsky27@gmail.com, anisurpm@gmail.com, sumathivel89@gmail.com

Abstract. We consider the mobile robot dispersion problem in the presence of faulty robots (crash-fault). Mobile robot dispersion consists of $k \leq n$ robots in an n-node anonymous graph. The goal is to ensure that regardless of the initial placement of the robots over the nodes, the final configuration consists of having at most one robot at each node. In a crash-fault setting, up to $f \leq k$ robots may fail by crashing arbitrarily and subsequently lose all the information stored at the robots, rendering them unable to communicate. In this paper, we solve the dispersion problem in a crash-fault setting by considering two different initial configurations: i) the rooted configuration, and ii) the arbitrary configuration. In the rooted case, all robots are placed together at a single node at the start. The arbitrary configuration is a general configuration (a.k.a. arbitrary configuration in the literature) where the robots are placed in some $l < k$ clusters arbitrarily across the graph. For the first case, we develop an algorithm solving dispersion in the presence of faulty robots in $O(k^2)$ rounds, which improves over the previous $O(f \cdot \min(m, k\Delta))$-round result by [23]. For the arbitrary configuration, we present an algorithm solving dispersion in $O((f + l) \cdot \min(m, k\Delta, k^2))$ rounds, when the number of edges m and the maximum degree Δ of the graph is known to the robots.

Keywords: Distributed algorithm · Mobile robot · Dispersion · Fault-tolerant algorithm · Crash-fault · Round complexity · Memory complexity

1 Introduction

The dispersion of autonomous mobile robots to spread them out evenly in a region is a problem of significant interest in distributed robotics, e.g. [9,10]. Initially, this problem was formulated by Augustine and Moses Jr. [2] in the context of graphs. They defined the problem as follows: Given any arbitrary initial configuration of $k \leq n$ robots positioned on the nodes of an n-node anonymous graph, the robots reposition autonomously to reach a configuration where each robot is positioned on a distinct node of the graph. Mobile robot dispersion has various real-world and practical applications, such as the relocation of self-driving electric cars (robots) to recharge stations (nodes). Assuming

The work of A.R. Molla and S. Sivasubramaniam were supported, in part, by ISI DCSW/TAC Project (file no. G5446 and G5719). A full version of the paper can be found at [4].

that the cars have smart devices to communicate with each other, the process to find a free or empty charging station, coordination including exploration (to visit each node of the graph in minimum possible time), scattering (spread out in an equidistant manner in symmetric graphs like rings), load balancing (nodes send or receives loads, and distributes them evenly among the nodes), covering, and self-deployment can all be explored as mobile robot dispersion problems [11,13–15].

The problem has been extensively studied in different graphs with varying assumptions since its conceptualization [11–16,19–22]. In this paper, we continue the study about the trade-off of memory requirement and time to solve the dispersion problem. Recently, Pattanayak et al. [23] explored the problem of dispersion in a set-up where some of these mobile robots are prone to crash faults. Whenever a robot crashes, it loses all its information immediately, as if the robot has vanished from the network. This makes the problem more challenging and also makes it more realistic in terms of real-world scenarios, where faulty robots can crash at any moment. In this paper, we have continued to study the efficacy of the problem in the same faulty environment. We studied the dispersion problem with both the rooted and arbitrary configuration of the robots in a faulty setup. Both algorithms maintain the optimal level of memory requirement for each robot.

The following table (Table 1) lists up the major notations used throughout the paper.

Table 1. List of major notations

Symbols	Meaning
G	The arbitrary graph acting as the underlying network for the robots
n	The number of nodes (vertices) in G
m	The number of edges in G
Δ	The highest degree of the nodes in G
k	Number of robots
f	Number of faulty robots among the k robots
l	Number of initial clusters of robots in the $arbitrary$ configuration
r_i	A robot with ID i
R_c	$root$ node in the $rooted$ configuration

1.1 Our Results

We consider a team of $k \leq n$ mobile robots placed on an arbitrary, undirected simple graph, consisting of n anonymous, memory-less nodes and m edges. The ports at each node are labelled. The robots have unique IDs and a restricted amount of memory (measured in the number of $bits$). These robots have some computing capability and can communicate with the other robots, only when they are at the same node. We consider two different starting scenarios, based on the initial configuration of the robots. When the robots start from a single node, we call the configuration as $rooted$, otherwise, we call it an $arbitrary$ configuration. We further assume that $f \leq k$ faulty robots in the network are prone to crash at any point of time. Our first algorithm for the rooted configuration crucially uses depth first search (DFS) traversal and improves the round complexity from $O(f \cdot \min(m, k\Delta))$ [23] rounds to $O(k^2)$. The second algorithm for the arbitrary configuration is an entirely new result whose complexity depends upon the factors: the number of faulty robots (f), number of robot clusters (l), total number of edges in the graph (m), number of robots (k) and the highest degree of the graph (Δ). In this case, the round complexity is $O((f + l) \cdot \min(m, k\Delta, k^2))$. The results are summarized in the following two theorems:

Theorem 1 (Crash Fault with Rooted Initial Configuration). *Consider any rooted initial configuration of $k \leq n$ mobile robots, out of which $f \leq k$ may crash, positioned on a single node of an arbitrary, anonymous n-node graph G having m edges, in synchronous setting* DISPERSION *can be solved deterministically in $O(k^2)$ rounds with $O(\log(k + \Delta))$ bits memory at each robot, where Δ is the highest degree of the graph.*

Theorem 1 improves over the previously known algorithm (in the worst case, improvement is from cubic to quadratic) that takes $O(f \cdot \min(m, k\Delta))$ rounds for f faulty robots [23]. The theorem also matches the optimal memory bound ($\Omega(\log(\max(k, \Delta)))$ [13]) with $O(\log(k+\Delta))$ bit memory and can handle any number of crashes.

Theorem 2 (Crash Fault with arbitrary Initial Configuration). *Consider any arbitrary initial configuration of $k \leq n$ mobile robots, out of which $f \leq k$ may crash and positioned on $l \leq k/2$ nodes of an arbitrary and anonymous n-node graph G having m edges, in synchronous setting* DISPERSION *can be solved deterministically in $O((f + l) \cdot \min(m, k\Delta, k^2))$ time with $O(\log(k + \Delta))$ bits memory at each robot.*

Theorem 2 solves the dispersion for arbitrary configuration with optimal memory per robot. The time complexity matches the one conjectured by Pattanayak *et al.* [23]. When f, l and Δ are constants, the time complexity matches the lower bound of $\Omega(k)$. Moreover, the algorithm can handle any number of faulty robots. The results are summarized in the Table 2.

2 Related Work

The problem of dispersion was first introduced in [2] by Moses Jr. *et al.*, where they solved the problem for different types of graphs. They had given a lower bound of $\Omega(\log n)$ on the memory of each robot (later, made more specific with $\Omega(\log(\max(k, \Delta)))$ in [13]) and of $\Omega(D)$ on the time complexity, for any deterministic algorithm on arbitrary graphs. They also proposed two algorithms on arbitrary graphs, one requiring $O(\log n)$ memory and running for $O(mn)$ time while the other needing a $O(n \log n)$ memory and having a time complexity of $O(m)$.

Kshemkalyani and Ali [11] provided several algorithms for both synchronous and asynchronous models. In the synchronous model, they solved the dispersion problem in $O(\min(m, k\Delta))$ rounds with $O(k \log \Delta)$ memory. For the asynchronous cases, they proposed several algorithms, one particularly requiring $O(\Delta^D)$ rounds and $O(D \log \Delta)$ memory, while another requiring $O(\max(\log k, \log \Delta))$ memory and having a time complexity of $O((m-n)k)$. In a later work, Kshemkalyani *et al.*, in [13] improved the time complexity to $O(\min(m, k\Delta)\log k)$ keeping the memory requirement to $O(\log n)$, while requiring that the robots know the parameters m, n, k, Δ beforehand. In a subsequent work, [24] kept the time and memory complexity of [13] intact while dropping the requirement of the robots having to know m, k, Δ beforehand. Recently, Kshemkalyani and Sharma [16] improved the time complexity to $O(\min(m, k\Delta))$. Works of [6,22] used randomization, which helped to reduce the memory requirement for each robot.

In [12], Kshemkalyani *et al.*, studied the problem in the *Global Communication Model*, in which the robots can communicate with each other irrespective of their positions in the graph[1]. The authors obtained a time complexity of $O(k\Delta)$ rounds when $O(\log(k+\Delta))$ bits of memory were allowed at each robot. Whereas, when robots were allowed $O(\Delta + \log k))$ bits, the number of rounds reduced to $O(\min(m, k\Delta))$. Both were for arbitrary initial configuration of robots. They also used BFS traversal techniques for investigating the dispersion problem. The BFS traversal technique yielded a time of $O((D+k)\Delta(D+\Delta))$ rounds with $O(\log D + \Delta \log k)$ bits of memory at each robot, using global communication, for arbitrary starting configuration of robots. Here D denotes the diameter of the graph. The problem was also studied on *dynamic* graphs in [1,15,17]. *Graph Exploration*, which is a related problem, has also been intensively studied in literature [3,5,7,8]

The dispersion problem has also been recently studied for configurations with faulty robots. In [18], Molla *et al.*, considered the problem for anonymous rings, tolerating weak Byzantine faults (robots that behave arbitrarily but cannot change their IDs). They gave three algorithms **(i)** the first one being memory optimized, requiring $O(\log n)$ bits of memory, $O(n^2)$ rounds and tolerating up-to $n-1$ faults.**(ii)** the second one is time optimized with $O(n)$ rounds, but require $O(n \log n)$ bits of memory, tolerating up-to $n-1$ faults. **(iii)** the third one runs in $O(n)$ time and $O(\log n)$ memory but cannot tolerate more than $[\frac{n-4}{17}]$ faulty robots. In [20], the authors proposed several algorithms for dispersion with some of them tolerating strong Byzantine robots (robots that behave arbitrarily and can tweak their IDs as well). Their algorithms are mainly based on the idea of gathering the robots at a root vertex, using them to construct an isomorphic map of G and finally dispersing them over G according to a specific protocol. However, their algorithms take exponential rounds for strong Byzantine robots starting from an arbitrary configuration. For the rooted configuration, their algorithm takes $O(n^3)$ rounds, but tolerates no more than $[n/4-1]$ strong Byzantine robots. Dispersion under *crash faults* has been dealt with in [23]. In [23], Pattanayak *et al.*, have considered the problem for a team of robots starting at a rooted configuration, with some robots being crash prone. Their algorithm handles an arbitrary number of crashes, with each robot requiring $O(\log(k+\Delta))$ bits of memory. The algorithm completes in $O(f \cdot \min(m, k\Delta))$ rounds. In our paper, we improve this time complexity while keeping the memory requirement to optimal and also extend the problem for the robots starting in arbitrary configuration. A comparison between our results and the most aligned works is shown in Table 2.

[1] In the *Local Communication Model* robots can communicate with each other only when they are at the same node.

Table 2. Results on Dispersion of $k \leq n$ robots with $f \leq k$ faulty robots on n-node arbitrary anonymous graphs having m edges such that Δ is the highest degree of the graph in the local communication model. Each uses an optimal memory of $O(\log(k + \Delta))$ bits on each robot.

Algorithm	Initial Config.	Crash handling	Time
Kshemkalyani *et al.* [16]*	Arbitrary	No	$O(\min(m, k\Delta))$
Pattanayak *et al.* [23]	Rooted	Yes	$O(f \cdot \min(m, k\Delta))$
Algorithm in Sect. 4	Rooted	Yes	$O(k^2)$
Algorithm in Sect. 5	Arbitrary	Yes	$O((f + l) \cdot \min(m, k\Delta, k^2))$

*The best known result as of now for fault-free dispersion.

3 Model

We now elaborate our model in detail.

Graph: The underline graph G is connected, undirected, unweighted and anonymous with n vertices and m edges. The vertices of G (also called nodes) do not have any distinguishing identifiers or labels. The nodes do not possess any memory and hence cannot store any information. The degree of a node $i \in V$ is denoted by δ_i and the maximum degree of G by Δ. Edges incident on i are locally labelled using a port number in the range $[1, \delta_i]$. A single edge connecting two nodes receives two independent port numbers at either end. The edges of the graph serve as $routes$ through which the robots can commute. Any number of $robots$ can travel through an edge at any given time.

Robots: We have a collection of $k \leq n$ robots $\mathbb{R} = \{r_1, r_2, ..., r_k\}$ residing on the nodes of the graph. Each robot has a unique ID and has some memory to store information. The robots cannot stay on an edge, but one or more robots can be present at a node at any point of time. A group of such robots at a node is called $co-located$ robots. Each robot knows the port number through which it has entered and exited a node.

Crash Faults: The robots are not fault-proof and a faulty robot can $crash$ at any time during the execution of the algorithm. Such $crashes$ are not recoverable and once a robot $crashes$ it immediately loses all the information stored in itself, as if it was not present at all. Further, a crashed robot is not visible or sensible to other robots. We assume there are f faulty robots such that $f \leq k$.

Communication Model: Our paper considers a local communication model where only the co-located robots can communicate among themselves.

Time Cycle: Each robot r_i, on activation, performs a $Communicate - Compute - Move$ (CCM) cycle as follows.

– Communicate: r_i reads its own memory along with the memory of other robots co-located at a node v_i.

– Compute: Based on the gathered information and subsequent computations, r_i decides on several parameters. This includes, deciding whether to settle at v_i or otherwise determine an appropriate exit port, choosing the information to pass/store at the settled robot and the information to carry along-with, if, exiting v_i.
– Move: r_i moves to the neighbouring node using the computed exit port.

We consider a synchronous system, where every robot is synchronized to a common clock and becomes active at each time cycle or round.

Time and Memory Complexity: We evaluate the time in terms of the number of discrete rounds or cycles before achieving DISPERSION. Memory is the number of bits of storage required by each robot to successfully execute DISPERSION. Our goal is to solve DISPERSION using optimal time and memory.

Let us now formally state the problem of fault-tolerant dispersion below.

Definition 1 (Fault-Tolerant Dispersion). *Given $k \leq n$ robots, up to f of which are faulty (which may fail by crashing), initially placed arbitrarily on a graph of n nodes, the non-faulty robots, i.e., the robots which are not yet crashed must re-position themselves autonomously to reach a configuration where each node has at most one (non-faulty) robot on it and subsequently terminate.*

4 Crash-Fault Dispersion for Rooted Configuration

In this section, we present a deterministic algorithm that disperses the robots with single-source (rooted configuration) in adaptive crash fault. Our goal is to minimize the round complexity as well as keep the memory of the robots low. The pseudocode and a pictorial description of the algorithm can be found in the full version of the paper [4].

4.1 Algorithm

In the absence of faulty nodes, one can run the DFS (depth first search) algorithm to solve the robot dispersion problem in $O(min(m, k\Delta))$ rounds. But in the presence of crash faults, due to crashes, it becomes challenging to explore the graph. Classic dispersion algorithms rely on the robots themselves to keep track of the paths during exploration. The presence of a crashed robot in this instance may lead to an endless cycle. Therefore, our goal is to ensure the dispersion of mobile robots despite the presence of faulty robots.

In the rooted configuration, to manage the presence of faults, we avoid exploring the graph together with all the robots. That is, the graph is explored sequentially such that each robot r_i ($1 \leq i \leq k$) does not begin exploring the graph, until the previous robot r_{i-1} is guaranteed to have settled. During exploration, whenever a robot r_i finds an empty node it settles down at that spot. Let us call this algorithm as ROOTED-CRASH-FAULT-DISPERSION. Below, we explain the algorithm in detail.

Functionality: For simplicity, let us assume that the robot's ID lies in the range of $[1,\ k]$. Otherwise, the robots can map their IDs from the actual range to the range $[1,\ k]$, since the IDs are distinct. We denote the rooted configuration by R_c. We slightly abuse notation and use R_c to indicate both the root and the initial gathering of robots. Robots at R_c traverse the graph via the DFS (Depth First Search) approach, where the decision of which edge to traverse first is based on the port numbers. The process proceeds in increasing order of IDs, starting with the robot with the minimum ID at R_c. R_c then sends each robot to explore the graph via DFS.

Let the robot with the current minimum ID be r_i. Then r_i begins to explore the graph via DFS (starting with the minimum port number at R_c). Once it leaves R_c, it has $3i$ rounds within which it can either i) settle at the first empty node it finds or ii) return to R_c if it does not find an empty node to settle within $2i$ rounds. If r_i reports to R_c within $3i$ rounds, then R_c ensures that it does not release the robot with the next lowest ID, say r_{i+1}. This can be guaranteed as r_i needs to traverse at most $(i-1)$ edges to explore the sub-graph traversed by r_{i-1}. r_i requires at most i rounds to return to the base R_c since the next traversed edge might lead to the already visited node which is not empty. As r_i requires i rounds to report at the R_c, therefore, r_i explores the graph for only $2i$ rounds. Notice that a robot will not traverse the distance of more than $(i+1)$, before that, there will be an empty edge at a distance (distance from the root) of $(i+1)$ and the robot will settle down there. If r_i did not find the empty node within $2i$ rounds then it starts to traverse towards R_c. In this way, r_i reports to R_c within $3i$ rounds so that R_c does not send another robot to explore the graph. R_c re-sends r_i to explore the graph. In this way, any r_i traverses the graph until it finds an empty node. Note that in our process, we ensure that there are no two robots that are exploring the graph at the same time.

To maintain the protocol, each r_i maintains the following fields. Its ID (r_i), a parent pointer $(r_i.parent)$ that represents the edge it traversed, a current direction pointer $(r_i.cdr)$ which indicates the direction it is required to follow. And finally, a backward traversal value $(r_i.B)$ which is initially 0, and is set to 1 once the backward traversal is complete. Here, our procedure performs the traditional DFS protocol but one-by-one, that is, the robots do not explore the graph simultaneously. A detailed account of the DFS traversal is provided in the full version [4].

Note that we have not addressed the case where a robot finds an empty node when returning to R_c (because the previously settled robot has crashed). In such an instance, the newly settled robot has a $r_i.parent$ and $r_i.cdr$ that point in the inappropriate direction. We address this condition below.

Decision: If r_i encounters an unexpected child, r_u i.e., a child whose parent and current pointer direction are set in the inappropriate direction w.r.t the perspective of r_i, it considers (correctly) that r_u replaced a robot that has previously crashed. In such a situation, r_i changes the parent of r_u appropriately, i.e., minimum available port number other than $r_u.parent$.

Lemma 1. *In the non-faulty setup, round complexity is* $O(k^2)$.

Proof. In a non-faulty setup, each robot behaves robustly and there are no crashes. Therefore, after the backtracking flag is set on a node, an edge is not traversed again during the DFS traversal. In traversing a graph from R_c, two kinds of situations may arise, either a robot r_i reaches an empty node after $O(i)$ edge traversals, or it traverses

$O(i^2)$ edges. In the first case, there is an empty node at a distance of $O(i)$. Therefore, r_i settles at the empty node after $O(i)$ rounds. If such kind of situation arises repeatedly, then the algorithm takes $O(1) + O(2) + \cdots + O(k) = O(k^2)$ rounds. In the second case, there might be a situation such that r_i traverses $O(i^2)$ edges to find the empty node and only encounters previously settled nodes (at most $i(i-1)/2$ edges). More preciously, $i/2$ new edges are traversed in $3i$ rounds. Notice that a robot will traverse only earlier traversed nodes at the distance $(i+1)$, if not, then there will be an empty edge at a distance (distance from the root) of $(i+1)$ and the robot will settle down there. Therefore, r_i covers $O(i^2)$ edges in $O(i^2)$ rounds and future robots (i.e., robots having ID $r_j; \forall\, j > i$) will not traverse these edges again. Hence, we can conclude that the non-faulty setup takes $O(k^2)$ rounds in the given model. □

Lemma 2. *In the faulty setting, a crashed robot may bring about an extra cost of $O(k)$ rounds in comparison to the non-faulty setting.*

Proof. In the faulty setup, a robot might crash at any time and the respective node becomes empty, say node v_i. As a consequence, the information held by that robot (at the node v_i) is also lost. Accordingly, the next robot that discovers v_i, say r_i, settles down at v_i. A robot possesses the information of current direction, parent node and backtracking status apart from its own ID. For that reason, the current direction pointer is pointing towards the edge based on its least labelled edge. But there might be the case (in the worst case) that the last crashed node has traversed up to $(i-2)$ edges which should be traversed again by the r_{i+1}. This takes extra $O(i)$ rounds. Also, in the worst case, this value can be $O(k)$ since the number of robots is k. Hence, the lemma. □

Lemma 3. *There is (at most) one robot moving (neither settled at its respective node, nor at rooted configuration R_c) at any instance.*

Proof. Proof by contradiction, let us suppose there exist two robots in moving condition, say r_i and r_{i+1}. Also, assume r_i started before, r_{i+1}. Now, as r_i has not settled, r_i reports to R_c every $3i$ rounds. But if r_i reports every $3i$ rounds then R_c does not release the next robot which is contradictory to our assumption. □

Lemma 4. *A loop or cycle may be formed by the current direction pointer (cdr pointer). The algorithm* ROOTED-CRASH-FAULT-DISPERSION *successfully avoids any loop during dispersion.*

Proof. During the execution of the algorithm, a loop or cycle may be formed if a robot r_i crashes at a node n_i then the current direction pointer ($cdr\ pointer$) is set by the upcoming robot r_{i+1} with the lowest port. That lowest port might have been traversed earlier. Therefore, a loop is formed. From Lemma 3, we know that only one robot is moving at any instance, say r_{i+1}. Therefore, r_{i+2} (the next robot) starts after r_{i+1} settles. If r_{i+2} encounters any robot with an unexpected $cdr\ pointer$ then r_{i+1} changes the $cdr\ pointer$ appropriately. Thus, loops are avoided in the network. □

Lemma 5. *The algorithm* ROOTED-CRASH-FAULT-DISPERSION *takes at most $7k^2$ rounds and $O(\log(k + \Delta))$ bits memory.*

Proof. In case of round complexity, a non-faulty set-up from Lemma 1, the total number of rounds are $3(1 + 2 + \cdots + k) < 3k^2$ (in the best case where r_i finds the empty node within $3i$ rounds). Additionally, a robot can traverse at most $i/2$ new edges in $3i$ rounds (in a particular phase) without settling down on an empty node (in the worst case). Therefore, round complexity for $k(k-1)/2$ edges in the non-faulty setup is $< 3k^2$. Moreover, from Lemma 2, we know that the extra cost incurred for f robot's crashing is at most fk. Hence, overall round complexity is at most $3k^2 + 3k^2 + k^2 = 7k^2$.

In case of memory complexity, each robot stores its ID which takes $O(\log k)$ bit space. Along with that parent pointer and current direction pointer takes $O(\log \Delta)$ bit memory each. While the backward pointer takes a single bit. Therefore, the memory complexity is $O(\log(k + \Delta))$. □

From the above discussion, we conclude the following result.

Theorem 1. *Consider any rooted initial configuration of $k \leq n$ mobile robots, out of which $f \leq k$ may crash, positioned on a single node of an arbitrary, anonymous n-node graph G having m edges, in synchronous setting* DISPERSION *can be solved deterministically in $O(k^2)$ time with $O(\log(k + \Delta))$ bits memory at each robot, where Δ is the highest degree of the graph.*

5 Crash-Fault Dispersion for Arbitrary Configurations

In this configuration setting, the robots are distributed across the graph in clusters such that there are $C = \{C_1, \ldots, C_l\}$ groups of robots at l different nodes at the start such that $\sum_i C_i = k$. The goal of the dispersion is to ensure that the robots are dispersed among the graph vertices such that each node has at most one robot. In this setting, we assume that the robots are aware of k, f, Δ, l and m.

Procedure: Our protocol runs in phases, in which each phase consists of min$(m, k\Delta, k^2)$ rounds. At the start of each phase, each cluster begins a *counter* that counts down from min$(m, k\Delta, k^2)$. Each cluster C_i then begins exploring the network simultaneously via the traditional DFS algorithm (in the trivial case of a singleton cluster consisting of only one robot, it considers itself dispersed). Unlike the rooted configuration, individual robots do not explore and return, the entire cluster moves together. Whenever a cluster encounters a new (empty) node in the network, the robot with the current highest ID in the cluster settles, and sets its pointers appropriately. At the end of each round, the counter is decreased by 1. When the counter becomes zero, it signals the end of the phase, and all flags are reset. That is all pointers become null, including the pointers of already settled robots. After that, each cluster starts exploring the network with its current node as a point of origin. This continues until all robots in the cluster settle. The pseudocode and a pictorial description of the algorithm can be found in the full version of the paper [4].

Detailed Procedure. There are two main parts to the protocol, i) exploration, ii) encounter. Exploration deals with the general procedure involved in exploring the graph, while encounter deals with the details involved when robots from different clusters meet.

Let's begin with all the information stored at a robot. Each robot r in a cluster C_i consists of the following pointers cid, $parent$, cdr, $priority$, and B (backtrack). The pointer cid denotes the ID of the cluster it belonged to when a robot settles. cid of a cluster C_i is determined at the start of the phase, and it is the ID of the robot with the highest ID. When a robot decides to settle at a node, the $parent$ pointer keeps track of the port through which it entered the node. Similarly, the cdr pointer is used to keep track of the port through which a cluster leaves the node in which it is settled. The $priority$ pointer of a settled robot keeps track of its priority in various clusters, originally this is simply the cid of the cluster it was part of, that is the priority of a cluster is simply its cid. However, a robot's priority may change if a higher priority cluster discovers it and updates its priority pointer. In our work, priority is decided by the cluster's ID, that is, between two clusters, the cluster with the higher cid has higher precedence. And of course, the backtrack pointer B keeps track of the backtrack status of its DFS. In addition to all of these, each robot also has a field called $counter$, which is set to $\min(m, k\Delta, k^2)$ at the beginning of each phase. Note that since all robots set the counter at the beginning of the phase simultaneously, the counter has the same value across all robots.

Exploration. As mentioned before, as long as a cluster is non-empty, at the beginning of each phase, each cluster begins exploring the graph via the traditional DFS until the cluster is empty or it encounters a robot from a higher priority cluster (more on this in the encounter section). In each phase, each robot in any cluster C_i sets its cid and $priority$ to the highest ID in the cluster, and its counter to $\min(m, k\Delta, k^2)$. We consider the node in which C_i is at the start of the phase, to be its root. C_i then follows the traditional DFS format for exploration. It leaves the node via the smallest unexplored port. If the node is empty, the robot with the current minimum ID, say r, sets its *parent* and cdr pointers and settles at the node.

The update function for the cdr pointer is exactly the same as the one in the rooted case, i.e., it follows the traditional DFS procedure, except that all the robots in the cluster move through the exit port. All robots in C_i decrease their counter by one and C_i leaves through the port in $r.cdr$. If a cluster ever finds itself returning to a node with a robot r from its own cluster and it has exhausted all of the ports in which r has settled, then it sets r's backtrack flag. Once a phase has finished, if the cluster is non-empty it resets all flags and counter and begins DFS once again. During exploration, if the cluster C_i reaches a node u whose degree is k, then they use BFS to explore the neighbourhood of u and settle the robots of C_i in at most $O(k)$ rounds. However, here we have not explored what happens if a robot from a cluster C_i meets a robot from C_j. That brings us to the next important part of the protocol, the encounters.

Encounter. This section contains the explanation of the encounter part of the protocol. When a robot (or cluster) meets, that is *encounters* a robot from a different cluster, the next step in exploration is decided based on priority. Simply put, the robot with higher priority always takes precedence as follows. There are two distinct scenarios, i) a cluster finds an already settled node or ii) multiple clusters meet on the same node. In the first case, if a cluster with a higher priority (say C_i) finds a robot r_p from a lower priority cluster (say C_j) on a node, it sets r_p's priority to its own priority, resets r_p's parent and cid to its own, and finally sets r_p's cdr (to the minimum unexplored port the higher priority cluster has not explored so far) and continues its DFS. If on the other hand, a lower priority cluster finds a robot r_p with a higher priority, it stops its exploration and just continues decreasing its counter at every round till the end of the phase, and begins the exploration in the next phase. Note, if a cluster finds a settled robot whose flags have been reset (i.e., set to $null$), then it's the same scenario as that of finding a robot from a lower priority cluster. The settled robot takes the priority and ID of the newly arrived cluster.

In the second scenario, if two (or more) clusters meet, the clusters merge and take on the priority of the cluster with the highest priority among them. They stop and count-down and begin exploration as a merged cluster in the next phase.

Note that the number of clusters is non-increasing between two consecutive phases. At any phase, a cluster may either (i) disperse over the nodes completely, or (ii) survive to explore in the next phase, or (iii) merge with a higher priority cluster. Thus, the number of clusters either remains the same or decreases at the end of every phase. Now we show that after $(l + f)$ phases, dispersion is achieved.

Lemma 6. *The effects of a robot crash, that is time delay caused by the presence of a crash are limited to the phase it occurs in. After that, it ceases to have effect.*

Proof. Since at the end of every phase, all robots reset their flags, including the parent and cdr pointers, previously explored paths are equivalent to new unexplored paths in the current phase, as their pointers are set by the currently exploring clusters. Hence, previous phases do not have any impact on the DFS running in the current phase. $\square$

Lemma 7. *Let C_i be the cluster with the highest priority in Phase j. C_i is guaranteed dispersion by the end of j if j is fault-free.*

Proof. From Lemma 6 we know that crashes in previous rounds do not have an effect on exploration in the current phase. And, in the absence of faults during the phase itself, we see that C_i exploration is equivalent to a rooted single cluster exploration of the network. Thus it is able to complete its dispersion using DFS without any delays or interference from other clusters, which takes less than $O(\min(m, k\Delta, k^2))$ rounds to complete. $\square$

Lemma 8. *Each cluster $C_i \in C$ is guaranteed to have at least a single fault-free phase in which it has the highest priority.*

Proof. Quite trivially, since there are $(l + f)$ phases, each cluster is guaranteed at least one phase in which no faults occur, and in which they are the highest priority. $\square$

Lemma 9. *At the end of $(l + f)$ phases, all clusters are guaranteed to have dispersed.*

Proof. This follows directly from Lemmas 7 and 8. Each cluster is guaranteed to have at least one fault-free phase in which it has the highest priority. From 6 we know, in that phase, there is guaranteed dispersion. Hence, in $(l + f)$ phases, we are guaranteed to have complete dispersion of all clusters. $\square$

Thus, we have the following theorem.

Theorem 2. *In the synchronous setting, the crash-tolerant algorithm for the arbitrary configuration (algorithm* Arbitrary-Crash-Fault-Dispersion*) ensures dispersion of mobile robots in an arbitrary graph from an arbitrary initial configuration in* $O((f+l) \cdot min(m, k\Delta, k^2))$ *rounds with each robot requiring* $O(\log(k+\Delta))$ *bits of memory.*

Remark 1. If only the number of robots (k) is known and all other factors are unknown to the network then the algorithm for arbitrary configuration takes $O(k^3)$ rounds.

6 Conclusion and Future Work

In this paper, we studied Dispersion for distinguishable mobile robots on anonymous port-labelled arbitrary graphs under crash faults. We presented a deterministic algorithm that solves robot dispersion in two different settings, i) a rooted configuration of robots and ii) an arbitrary configuration of robots. We achieved the $O(k^2)$ round complexity in rooted configuration while $O((f+l)\min(m, k\Delta, k^2))$ round complexity in arbitrary setting. In both cases, we used $O(\log(k+\Delta))$ bits of memory. Some open questions that are raised by our work: i) What is the non-trivial lower bound for the round complexity in both the setting by keeping the memory to $O(\log(k+\Delta))$. ii) If it is possible to give a similar round complexity for the case of arbitrary configuration as we achieved in rooted configuration. iii) If it is possible to get the same bound in the arbitrary configuration without the knowledge of f, l, Δ and m. iv) Finally, whether similar bounds hold in the presence of Byzantine failures.

References

1. Agarwalla, A., Augustine, J., Moses Jr., W.K., Madhav K.S., Sridhar, A.K.: Deterministic dispersion of mobile robots in dynamic rings. In: ICDCN (2018)
2. Augustine, J., Moses Jr., W.K.: Dispersion of mobile robots: a study of memory-time trade-offs. In: ICDCN (2018)
3. Bampas, E., Gasieniec, L., Hanusse, N., Ilcinkas, D., Klasing, R., Kosowski, A.: Euler Tour Lock-in Problem in the Rotor-Router Model. In: DISC (2009)
4. Chand, P.K., Kumar, M., Molla, A.R., Sivasubramaniam, S.: Fault-tolerant dispersion of mobile robots (2022). https://arxiv.org/abs/2211.12451
5. Cohen, R., Fraigniaud, P., Ilcinkas, D., Korman, A., Peleg, D.: Label-guided graph exploration by a finite automaton. ACM Trans, Algorithms **4**, 42 (2008)
6. Das, A., Bose, K., Sau, B.: Memory optimal dispersion by anonymous mobile robots. In: CALDAM (2021)
7. Dereniowski, D., Disser, Y., Kosowski, A., Pajak, D., Uznanski, P.: Fast collaborative graph exploration. In: ICALP (2013)
8. Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph Exploration by a Finite Automaton. Theorr. Comput. Sci. **345**, 331–344 (2005)
9. Hsiang, T., Arkin, E.M., Bender, M.A., Fekete, S.P., Mitchell, J.S.B.: Algorithms for rapidly dispersing robot swarms in unknown environments. In: WAFR (2002)

10. Hsiang, T., Arkin, E.M., Bender, M.A., Fekete, S.P., Mitchell, J.S.B.: Online dispersion algorithms for swarms of robots. In: Fortune, S. (ed.) SCG 2003
11. Kshemkalyani, A.D., Ali, F.: Efficient dispersion of mobile robots on graphs. In: ICDCN (2019)
12. Kshemkalyani, A.D., Molla, A.R., Sharma, G.: Dispersion of mobile robots on grids. In: WALCOM (2020)
13. Kshemkalyani, A.D., Molla, A.R., Sharma, G.: Fast dispersion of mobile robots on arbitrary graphs. In: ALGOSENSORS (2019)
14. Kshemkalyani, A.D., Molla, A.R., Sharma, G.: Dispersion of mobile robots using global communication. J. Parallel Distributed Comput. **121**, 100–117 (2022)
15. Kshemkalyani, A.D., Rahaman Molla, A., Sharma, G.: Efficient dispersion of mobile robots on dynamic graphs. In: ICDCS (2020)
16. Kshemkalyani, A.D., Sharma, G.: Near-optimal dispersion on arbitrary anonymous graphs. In: OPODIS (2021)
17. Luna, G.A.D., Flocchini, P., Pagli, L., Prencipe, G., Santoro, N., Viglietta, G.: Gathering in dynamic rings, pp. 79–98. Theor. Comput, Sci. **811**, 79–98 (2020)
18. Mandal, S., Molla, A.R., Jr., W.K.M.: Live exploration with mobile robots in a dynamic ring, revisited. In: ALGOSENSORS (2020)
19. Molla, A.R., Jr., W.K.M.: Dispersion of mobile robots. In: ICDCN (2022)
20. Molla, A.R., Mondal, K., Moses Jr., W.K.: Byzantine dispersion on graphs. In: IPDPS (2021)
21. Molla, A.R., Mondal, K., Moses Jr., W.K.: Efficient dispersion on an anonymous ring in the presence of weak byzantine robots. In: ALGOSENSORS (2020)
22. Molla, A.R., Moses Jr., W.K.: Dispersion of mobile robots: The power of randomness. In: TAMC (2019)
23. Pattanayak, D., Sharma, G., Mandal, P.S.: Dispersion of mobile robots tolerating faults. In: ICDCN (2021)
24. Shintaku, T., Sudo, Y., Kakugawa, H., Masuzawa, T.: Efficient dispersion of mobile agents without global knowledge. In: SSS (2020)