Space-efficient uniform deployment of mobile agents in asynchronous unidirectional rings

doi: https://doi.org/10.1016/j.tcs.2019.12.031

Space-efficient Uniform Deployment of Mobile Agents in Asynchronous Unidirectional Rings[☆,☆☆]

Masahiro Shibata[a,*], Hirotsugu Kakugawa[b], Toshimitsu Masuzawa[c]

[a]*Department of Computer Science and Electronics, Kyushu Institute of Technology 680-4 Kawazu, Iizuka, Fukuoka 820-8502, Japan*
[b]*Faculty of Science and Technology, Ryukoku University 1-5 Seta, Otsu, Shiga, 520-2194, Japan*
[c]*Graduate School of Information Science and Technology, Osaka University 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan*

Abstract

In this paper, we consider the uniform deployment problem of mobile agents in asynchronous unidirectional ring networks. This problem requires agents to spread uniformly in the network. In this paper, we focus on the memory space per agent required to solve the problem. We consider two problem settings. The first setting assumes that agents have no multiplicity detection, that is, agents cannot detect whether another agent is staying at the same node or not. In this case, we show that each agent requires $\Omega(\log n)$ memory space to solve the problem, where n is the number of nodes. In addition, we propose an algorithm to solve the problem with $O(k + \log n)$ memory space per agent, where k is the number of agents. The second setting assumes that each agent is equipped with the weak multiplicity detection, that is, agents can detect whether another agent is staying at the same node or not, but cannot get any other information about the number of the agents. Then, we show that the memory space per agent can be reduced to $O(\log k +$

[☆]The conference version of this paper is published in the proceedings of 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2018).

[☆☆]This work was partially supported by JSPS KAKENHI Grant Number 16K00018, 17K19977, and 18K18031, and Japan Science and Technology Agency (JST) SICORP.

[*]Corresponding author. Tel.:+81-9-4829-7656.

Email addresses: `shibata@cse.kyutech.ac.jp` (Masahiro Shibata), `kakugawa@rins.ryukoku.ac.jp` (Hirotsugu Kakugawa), `masuzawa@ist.osaka-u.ac.jp` (Toshimitsu Masuzawa)

$\log \log n$). To the best of our knowledge, this is the first research considering the effect of the multiplicity detection on memory space required to solve problems.

Keywords: distributed system, mobile agent, uniform deployment, ring network, space-efficient

1. Introduction

1.1. Background and related works

A *distributed system* consists of a set of computers (*nodes*) connected by communication links. As a promising design paradigm of distributed systems, (mobile) agents have attracted much attention [1, 2]. Agents can traverse the system with carrying information collected at visited nodes and process tasks on each node using the information. In other words, agents can encapsulate the process code and data, which simplifies design of distributed systems [3, 4].

In this paper, we consider the *uniform deployment* (or *uniform scattering*) *problem* as a fundamental problem for coordination of agents. This problem requires agents to spread uniformly in the network. Uniform deployment is useful for network management. In a distributed system, it is necessary that each node regularly gets software updates and is checked whether some application installed on the node is running correctly or not [5, 6]. Hence, considering agents with such services, uniform deployment guarantees that agents visit each node at short intervals and provide services. Uniform deployment might be useful also for a kind of load balancing. That is, considering agents with large-size database replicas, uniform deployment guarantees that not all nodes need to store the database but each node can quickly access the database [7, 8]. Hence, we can see the uniform deployment problem as a kind of the resource allocation problem (e.g., the k-server problem).

As related works, Flocchini et al. [9] and Elor et al. [10] considered uniform deployment with termination detection in ring networks, and Barriere et al. [11] considered uniform deployment with termination detection in grid networks. All of them assumed that agents are oblivious (or memoryless) but can observe multiple nodes within its visibility range. This assumption is often called the *Look-Compute-Move* model. On the other hand, our previous work [12] considered uniform deployment in asynchronous unidirectional ring networks for agents that have memory but cannot observe nodes except for

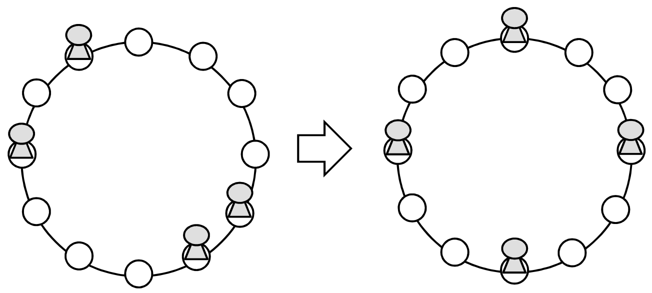


Figure 1: An example of uniform deployment ($n = 12, k = 4$)

their currently visiting nodes. We considered two problem settings: agents with knowledge of k and agents without knowledge of k, where k is the number of agents. For the first (resp., second) model, we proposed two (resp., one) algorithms to solve the uniform deployment problem with (resp., without) termination detection. Recently, a variation of the uniform deployment problem, called the *dispersion problem*, is considered in [13, 14]. In this problem, there exist n agents in an n-node graph, and this problem requires to deploy the agents so that each node is occupied by exactly one agent. Hence, we can see the dispersion problem as a special case of the uniform deployment problem when the number of agents is equal to the number of nodes. Note that while all of the above works for uniform deployment [9, 10, 11, 12] assumed that all agents are placed at distinct nodes in the initial configuration, the dispersion problem allows initial configurations such that several agents share the same node. So far, Kshemkalyani et al. [13] considered the dispersion problem with termination detection in arbitrary static networks and Agarwalla et al. [14] considered the problem with termination detection in dynamic rings.

1.2. Our contribution

In this paper, we consider the uniform deployment problem in asynchronous unidirectional ring networks (Fig. 1). Although we consider unidirectional rings, this result can be applied to most cases of bidirectional rings. If agents have a common sense of direction in the bidirectional ring, the result in unidirectional rings can be directly applied to bidirectional rings. If agents do not have a common sense of direction and the initial deployment of agents is symmetric, it is possible that agents have a common sense of

Table 1: Results for agents with knowledge of k (n: #nodes, $\log N$: upper bound of $\log n$, k: #agents)

	Previous results [12]		Results of this paper	
	Result 1	Result 2	Model 1	Model 2
Available Knowledge	k	k	k	$k, \log N$
Communication	Messages	Messages	Unremovable tokens	Unremovable tokens
Multiplicity detection	Required	Required	Not Required	Required
Termination detection	Required	Required	Required	Not Required
Agent memory	$O(k \log n)$	$O(\log n)$	$O(k + \log n)$	$O(\log k + \log \log n)$
Time complexity	$\Theta(n)$	$O(n \log k)$	$O(n \log k)$	$O(n^2 \log n)$
Total number of moves	$\Theta(kn)$	$\Theta(kn)$	$O(kn \log k)$	$O(kn^2 \log n)$

direction, and thus, the idea in unidirectional rings can be applied. Thus, considering the problem in unidirectional rings is fundamental one and interesting to investigate. Similar to [12], we assume that agents have memory but cannot observe nodes except for their currently visiting nodes. While the previous work [12] considered uniform deployment with such agents for the first time and clarified the solvability, this work focuses on the memory space per agent required to solve the problem and aims to propose space-efficient algorithms in weaker models than that of [12]. That is, while agents in [12] can send a message to the agents staying at the same node, agents in this paper do not have such ability. Instead, each agent initially has a token and can release it on a visited node, and agents can communicate only by the tokens. After a token is released, it cannot be removed. We also analyze the time complexity and the total number of moves.

In Table 1, we compare our contributions with the results for agents with knowledge of k in [12]. We consider two problem settings. The first setting considers agents *without* multiplicity detection, that is, agents cannot detect whether another agent is staying at the same node or not. In this model, we show that each agent requires $\Omega(\log n)$ memory space to solve the problem, where n is the number of nodes. In addition, we propose an algorithm to solve the problem *with* termination detection and this algorithm requires $O(k + \log n)$ memory space per agent, $O(n \log k)$ time, and $O(kn \log k)$ total number of moves. Note that, in the asynchronous system the (ideal) time complexity is defined under the following assumptions: 1) The time for an agent to transit to the next node is at most one, and 2) the time for local computation is ignored. The second setting considers agents *with* the weak multiplicity detection, that is, agents can detect whether another agent is staying at the same node or not, but they cannot get any other information about the number of the agents. In this setting, we also assume that agents know an upper bound $\log N$ of $\log n$ such that $\log N = O(\log n)$. Then, we propose an algorithm to solve the problem *without* termination detection and this algorithm reduces the memory space per agent to $O(\log k + \log\log n)$, but uses $O(n^2 \log n)$ time and $O(kn^2 \log n)$ total number of moves. To the best of our knowledge, this is the first research considering the effect of the multiplicity detection on memory space required to solve problems.

1.3. Organization

The paper is organized as follows. Section 2 presents the system model and introduces the uniform deployment problem. In Section 3 we consider the case without multiplicity detection. In Section 4 we consider the case with weak multiplicity detection. Section 5 concludes the paper.

2. Preliminaries

2.1. System model

We use almost the same model as that in [12]. A *unidirectional ring network* R is defined as 2-tuple $R = (V, E)$, where V is a set of anonymous nodes and E is a set of unidirectional links. We denote by $n\,(= |V|)$ the number of nodes, and let $V = \{v_0, v_1, \ldots, v_{n-1}\}$ and $E = \{e_0, e_1, \ldots, e_{n-1}\}$ $(e_i = (v_i, v_{(i+1) \bmod n}))$. For simplicity, operations on an index of a node assume calculations modulo n, that is, $v_{(i+1) \bmod n}$ is simply represented by v_{i+1}. We define the direction from v_i to v_{i+1} as the *forward* direction. The *distance* from node v_i to v_j is defined to be $(j - i) \bmod n$.

An agent is a state machine having an *initial state*. Let $A = \{a_0, a_1, \ldots, a_{k-1}\}$ be a set of $k\,(\leq n)$ anonymous agents. Since the ring is unidirectional, agents staying at v_i can move only to v_{i+1}. We assume that agents have knowledge of k but do not have knowledge of n. Each agent initially has a single *token* and can release it on a visited node at most once. After a token is released, it cannot be removed. The token on an agent can be realized by one bit memory and cannot carry any additional information. Hence, the tokens on a node represents just the number of the tokens and agents cannot recognize the owners of the tokens. This information is used to recognize whether the currently visited node is where an agent is initially located or not, and to detect when an agent has traveled once around the ring by comparing the number of tokens that it has observed with the number k of agents. Thus, it is sufficient that each agent has only one token. Notice that each node can store information other than tokens in practice, but it is sufficient to store information about tokens when considering anonymous agents. Moreover, we assume that agents move through a link in a FIFO manner, that is, when agent a_p leaves v_i after agent a_q, a_p reaches v_{i+1} after a_q. Note that such a FIFO assumption is natural because 1) agents are implemented as messages in practice, and 2) the FIFO assumption of messages is natural and can be easily realized using sequence numbers.

We consider two problem settings: agents *without multiplicity detection* and agents *with weak multiplicity detection*. While agents without multiplicity detection cannot detect whether another agent is staying at the same node or not, agents with weak multiplicity detection can detect another agent staying at the same node, but they cannot get any other information about the number of the agents. Hence, such multiplicity detection is called *weak*. Each agent can be equipped with the ability of weak multiplicity detection using only 1-bit memory space. Notice that in practice it requires little cost (at most $O(\log k)$ memory space per agent) to equip agents with the ability to count the exact number of agents staying at the same node (this ability is called the *strong multiplicity detection*). However, in this paper we aim to reduce memory space as much as possible, and thus we consider agents without (or with weak) multiplicity detection. In addition, it is possible that the ability of strong multiplicity detection does not work correctly for some fault. Even in this case, agents without (or with weak) multiplicity detection can work correctly. Thus, such agents achieve some kind of fault-tolerance.

Each agent a_i executes the following three operations in an atomic action: 1) Agent a_i reaches a node v (when a_i is in transit to v), or starts operations at v (when a_i stays at v), 2) agent a_i executes local computation, and 3) agent a_i leaves v if it decides to move. For the case with weak multiplicity detection, the local computation depends on whether another agent is staying at v or not. Note that these assumptions of atomic actions are also natural because they can be implemented by nodes with an incoming *buffer* that stores agents about to visit the node and makes them execute actions in a FIFO order. We consider an *asynchronous* system, that is, the time for each agent to transit to the next node or to wait until the next activation (when staying at a node) is finite but unbounded.

A (global) *configuration* C is defined as a 4-tuple $C = (S, T, P, Q)$ and the correspondence table is given in Table 2. Element S is a k-tuple $S =$

Table 2: Meaning of each element in configuration $C = (S, T, P, Q)$

Element	Meaning and example
$S = (s_0, s_1, \ldots, s_{k-1})$	Agent states (s_i: the state of agent a_i)
$T = (t_0, t_1, \ldots, t_{n-1})$	Node states (t_i: the number of tokens at node v_i)
$P = (p_0, p_1, \ldots, p_{n-1})$	Sets of agents staying at nodes (p_i: a set of agents staying at node v_i)
$Q = (q_0, q_1, \ldots, q_{n-1})$	Sequences of agents residing on links (q_i: a sequence of agents in transit from v_{i-1} to v_i)

$(s_0, s_1, \ldots, s_{k-1})$, where s_i is the state (including the state to denote whether it holds a token or not) of agent a_i. Element T is an n-tuple $T = (t_0, t_1, \ldots, t_{n-1})$, where t_i is the state (i.e., the number of tokens) of node v_i. The remaining elements P and Q represent the positions of agents. Element P is an n-tuple $P = (p_0, p_1, \ldots, p_{n-1})$, where p_i is a set of agents staying at node v_i. Element Q is an n-tuple $Q = (q_0, q_1, \ldots, q_{n-1})$, where q_i is a sequence of agents residing in the FIFO queue corresponding to link (v_{i-1}, v_i). Hence, agents in q_i are in transit from v_{i-1} to v_i.

We denote by $\mathcal{C}$ the set of all possible configurations. In *initial configuration* $C_0 \in \mathcal{C}$, all agents are in the initial state (where each has a token) and placed at distinct nodes and no node has any token. Notice that we assume such an initial configuration just for simplicity, but even if two or more agents exist at the same node in C_0, agents can solve the problem similarly by using the number of tokens at each node and atomicity of execution. The node where agent a is located in C_0 is called the *home node* of a and is denoted by $v_{HOME}(a)$. For convenience, we assume that in C_0 agent a is stored at the incoming buffer of its home node $v_{HOME}(a)$. This assures that agent a starts the algorithm at $v_{HOME}(a)$ before any other agents make actions at $v_{HOME}(a)$, that is, a is the first agent that takes an action at $v_{HOME}(a)$.

A (sequential) *schedule* $X = \rho_0, \rho_1, \ldots$ is an infinite sequence of agents, intuitively which activates agents to execute their actions one by one. Schedule X is *fair* if every agent appears in X infinitely often. An infinite sequence of configurations $E = C_0, C_1, \ldots$ is called an *execution* from C_0 if there exists a fair schedule $X = \rho_0, \rho_1, \ldots$ that satisfies the following conditions for each h ($h > 0$):

- If $\rho_{h-1} \in p_i$ (i.e., ρ_{h-1} is an agent staying at v_i) for some i in C_{h-1}, the states of ρ_{h-1} and v_i in C_{h-1} are changed in C_h by local computation of ρ_{h-1}. If ρ_{h-1} releases its token at v_i, the value of t_i increases by one. After this, if ρ_{h-1} decides to move to v_{i+1}, ρ_{h-1} is removed from p_i and is appended to the tail of q_{i+1}. If ρ_{h-1} decides to stay, ρ_{h-1} remains in p_i. The other elements in C_h are the same as those in C_{h-1}.

- If agent ρ_{h-1} is at the head of q_i (i.e., ρ_{h-1} is the next agent to reach v_i) for some i in C_{h-1}, ρ_{h-1} is removed from q_i and reaches v_i. Then, the states of ρ_{h-1} and v_i in C_{h-1} are changed in C_h by local computation of ρ_{h-1}. If ρ_{h-1} releases its token at v_i, the value of t_i increases by one. After this, if ρ_{h-1} decides to move to v_{i+1}, ρ_{h-1} is appended to the tail of q_{i+1}. If ρ_{h-1} decides to stay, ρ_{h-1} is inserted in p_i. The other elements in C_h are the same as those in C_{h-1}.

Note that if the activated agent ρ_{h-1} has no action to execute, then C_{h-1} and C_h are identical. Actually after uniform deployment is achieved, the same configuration is repeated forever.

2.2. The uniform deployment problem

The uniform deployment problem in a ring network requires $k\,(\geq 2)$ agents to spread uniformly in the ring, that is, all the agents are located at distinct nodes and the distance between any two *adjacent agents* should be identical like Fig. 1. Here, we say two agents are adjacent when there exists no agent between them. However, we should consider the case that n is not a multiple of k. In this case, we aim to distribute the agents so that the distance of any two adjacent agents should be $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$.

We consider the uniform deployment problem in two settings: *with termination detection* and *without termination detection*. In the uniform deployment problem with termination detection, a unique *halt state* is defined as follows. An agent stays at a node (not in a link) when it is in the halt state. When agent a_i enters the halt state, it terminates the algorithm, that is, a_i neither changes its state nor leaves the current node once it enters the halt state. Hence if an agent enters the halt state, it can detect its termination. Now, we define the uniform deployment problem with termination detection as follows.

Definition 1. *An algorithm solves the uniform deployment problem* with *termination detection if any execution satisfies the following conditions.*

- *All agents change their states to the halt state in finite time.*
- *When all agents are in the halt state, $q_j = \emptyset$ holds for any $q_j \in Q$ and the distance of each pair of adjacent agents is $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$.*

On the other hand, the uniform deployment problem without termination detection introduces *suspended states* instead of the halt state as follows. An agent stays at a node when it is in a suspended state. When agent a_i enters a suspended state, it neither changes its state nor leaves the current node v unless the observable local configuration of v (i.e., existence of another agent or the number of tokens for agents with weak multiplicity detection, or the number of tokens for agents without multiplicity detection) changes. If the local configuration changes, it can resume its behavior and leave the current node. Different from the halt state, the suspended states are not uniquely defined since an agent can resume different behaviors from different suspended states; the suspended state should contain the information necessary to resume the behavior. The uniform deployment problem without termination detection allows all agents to stop in the suspended states, which is also known as communication deadlock.

Definition 2. *An algorithm solves the uniform deployment problem* without *termination detection if any execution satisfies the following conditions.*

- *All agents change their states to the suspended states in finite time.*
- *When all agents are in the suspended states, $q_j = \emptyset$ holds for any $q_j \in Q$, and the distance of each pair of adjacent agents satisfies $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$.*

Next, we define the *time complexity* as the time required to solve the problem. Since there is no bound on time between activations in asynchronous systems, it is impossible to measure the exact time. Instead we consider the *ideal time complexity*, which is defined as the execution time under the following assumptions: 1) The time for an agent to transit to the next node or to wait until the next activation is at most one, and 2) the time for local computation is ignored (i.e., zero). This definition is based on the ideal time complexity for asynchronous message-passing systems [15]. For example, if some agent continues to move in the ring from the beginning to the end of execution of the algorithm, the ideal time complexity is equivalent to the number of moves for the agent. Note that these assumptions are introduced only to evaluate the complexity, that is, algorithms are required to work correctly without such assumptions. In the following, we use terms "time complexity" and "time" instead of "ideal time complexity".

3. Agents without multiplicity detection

In this section, we consider uniform deployment for agents without multiplicity detection.

3.1. A lower bound of memory space per agent

First, we show the following lower bound of memory space per agent.

Theorem 1. *When $k < n/2$ and $k = O(n^c)$ hold for any constant $c < 1$, for agents without multiplicity detection the memory space per agent to solve the uniform deployment problem is $\Omega(\log n)$ even if the algorithm does not require termination detection.*

Proof. We show the theorem by contradiction. We assume that there exists an algorithm to solve the uniform deployment problem with at most $\log n - \log 2k$ bit memory per agent. Notice that $\log n - \log 2k = \Omega(\log n)$ holds because $k < n/2$ and $k = O(n^c)$ hold for any constant $c < 1$. Then, each agent has at most $2^{(\log n - \log 2k)} = n/2k$ states. Since we consider a deterministic algorithm and an agent requires d states to store a distance of length d, when an agent enters the halt state or a suspended state, it has moved at most $n/2k$ times after it observed a token for the last time.

We consider the initial configuration such that agents $a_0, a_1, \ldots, a_{k-1}$ are placed at consecutive nodes in an n-node ring in this order. Then, the distance between two adjacent agents in the final configuration should be $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$. We assume that agents move in a synchronous manner, that is, in each step all agents are always activated and execute actions simultaneously. Then, since all agents execute the same deterministic algorithm and recognize the same local configuration until they release tokens, they release tokens simultaneously (if they do). Since agents stay at consecutive nodes in the initial configuration, the released tokens are also placed at consecutive nodes. We consider the behavior of agents a_0 and a_1. Since a_0 and a_1 move at most $n/2k$ times after they observed a token for the last time, the distance between them is at most $n/2k + 1 (\neq \lfloor n/k \rfloor$ or $\lceil n/k \rceil)$. However, this contradicts the condition of uniform deployment and we have the theorem. □

Remark. We proved the lower bound for any k such that $k = O(n^c)$ for the case that agents have knowledge of k. Of course, we can use the same proof idea for some specific values of k (e.g., $k = n/2$) even when k does not satisfy $k = O(n^c)$.

3.2. An algorithm with $O(k + \log n)$ memory space per agent

Next, we propose an algorithm to solve the uniform deployment problem with termination detection and this algorithm requires $O(k+\log n)$ memory space per agent, $O(n \log k)$ time, and $O(kn \log k)$ total number of moves. From Theorem 1, the algorithm is optimal in memory space per agent when $k = O(\log n)$. The algorithm consists of two phases as do the two algorithms in [12]: the selection phase and the deployment phase. In the selection phase, agents select some *base nodes*, which are the reference nodes for uniform deployment. In the deployment phase, based on the base nodes, each agent determines a *destination node* where it should stay and moves to the node. For simplicity, we assume $n = ck$ for some positive integer c, and we will remove this assumption in Section 3.2.3.

3.2.1. Selection Phase

In this phase, some home nodes are selected as base nodes. We say that two base nodes are *adjacent* when there exists no base node between them. Then, several base nodes are selected to satisfy the following conditions called the **base node conditions**: 1) At least one base node exists, 2) the distance between every pair of adjacent base nodes is the same, and 3) the number of home nodes between every pair of adjacent base nodes is the same. For example, in Fig. 2 (a) distances from $v_{HOME}(a_1)$ to $v_{HOME}(a_2)$, from $v_{HOME}(a_2)$ to $v_{HOME}(a_3)$, and from $v_{HOME}(a_3)$ to $v_{HOME}(a_1)$ are all 6, and the number of home nodes between $v_{HOME}(a_1)$ and $v_{HOME}(a_2)$, between $v_{HOME}(a_2)$ and $v_{HOME}(a_3)$, and between $v_{HOME}(a_3)$ and $v_{HOME}(a_1)$ are all 2. Thus, $v_{HOME}(a_1)$, $v_{HOME}(a_2)$, and $v_{HOME}(a_3)$ satisfy the base node conditions. Notice that it is possible that only a single node is selected as the base node. After selection of base nodes, agents that are not staying at the base nodes move so that they stay evenly between the base nodes and agents achieve uniform deployment (Fig. 2 (b)). When the selection phase is completed, each agent stays at its home node and knows whether its home node is selected as a base node or not. We call an agent a *leader* (but possibly not unique) when its home node is selected as a base node, and call it a *follower* otherwise. Active agents are candidates for leaders, and initially all agents are active.

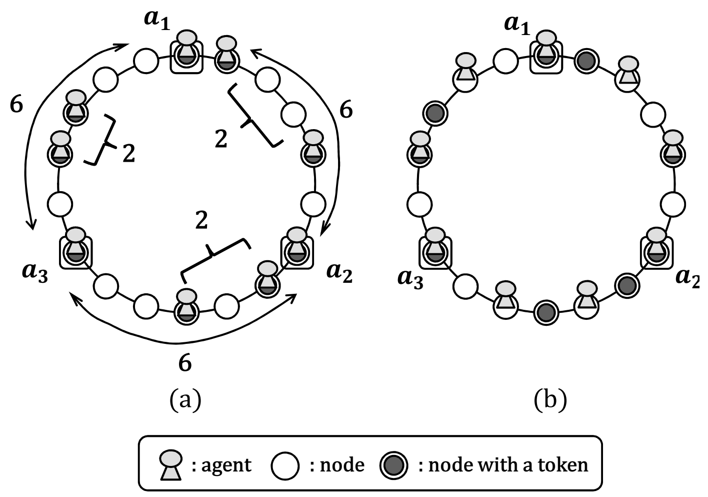


Figure 2: (a): An example of the base node conditions ($n = 18, k = 9$), (b): Behavior after selection of base nodes.

We say that node v is active (resp., a follower) when it is the home node of an active (resp., a follower) agent.

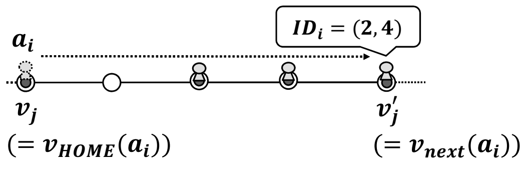
At first, we explain the basic idea of the selection phase in [12], which assumes weak multiplicity detection, and then we explain the way of applying the idea to the model in this section (i.e., the case without multiplicity detection). In the selection phase of [12], agents find *group IDs* (GID) (but possibly not unique) each of which is a tuple of the distance and the number of followers between two base nodes, and decrease the number of active agents using the GIDs. At the beginning of the algorithm, each agent a_i releases its token at $v_{HOME}(a_i)$. The selection phase consists of several subphases. At the beginning of each subphase, each agent a_i stays at $v_{HOME}(a_i)$. During the subphase, if a_i is a follower, it keeps staying at $v_{HOME}(a_i)$. On the other hand, each active agent a_i travels once around the ring and gets its new GID. Notice that each agent can detect when it completes one circuit of the ring using knowledge of k. Then, a_i compares its GID with GIDs of other agents one by one in a lexicographical manner (a_i gets them during the traversal of the ring) and determines the next behavior. Briefly, (a) if all active agents have the same GID, it means that home nodes of the active agents satisfy the base node conditions. Hence, the active agents become leaders and enter to the deployment phase. (b) If all agents do not have the same GID but a_i's GID is the maximum, it remains active and executes the next subphase. (c)

If a_i does not satisfy (a) or (b), it becomes a follower. Agents execute such subphases until base nodes are selected.

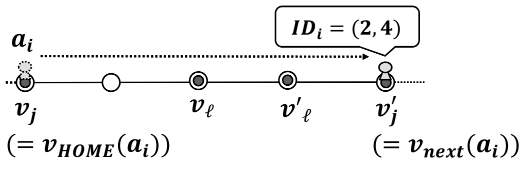
Now, we explain the detail of the group ID (GID). The GID (not necessarily unique) of an active agent a_i is given in the form of $(fNum_i, d_i)$, where $fNum_i$ is the number of follower nodes between $v_{HOME}(a_i)$ and the next active node in the subphase, say $v_{next}(a_i)$, and d_i is the distance from $v_{HOME}(a_i)$ to $v_{next}(a_i)$. In Fig. 3 (a), when agent a_i moves from its home node $v_j(= v_{HOME}(a_i))$ to the next active node $v'_j(= v_{next}(a_i))$, it observes two follower nodes and visits four nodes. Hence, a_i gets its GID $GID_i = (2, 4)$. Note that active agents traverse the ring and follower agents stay at their home nodes, and each active agent a_i executes the following three operations in one atomic action: 1) Agent a_i visits some node v, 2) agent a_i executes some computation at v, and 3) agent a_i leaves the current node. By the above fact and the FIFO property of links, it does not happen that some active agent visits some node where another active agent is still staying. Thus, each active agent a_i can detect its arrival at the next active node when it visits a token node with no agent (recall that the weak multiplicity detection is assumed in [12]). By the similar way, agent a_i can get the GIDs of every active agent while it travels once around the ring. Agents in [12] use $O(\log n)$ memory space to get such a GID and decide whether they remain active (or they have the lexicographically maximum GID) or not. Notice that an agent may get different GIDs in different subphases.

In the following, we explain how to apply the previous idea to the model in this section (i.e., the case without multiplicity detection). Agents in this section cannot detect existence of other agents staying at the same node and cannot detect the arrival of the next active node using existence of an agent. To deal with this, unlike [12], all agents (including follower agents) move in the ring and memorize the state of all agents by using a k-bits array $Active_{now}$. The value of $Active_{now}[i]$ is *true* iff its i-th agent is active (otherwise it is a follower). Hence, each agent can get a GID of any active agent a by going from node $v_{HOME}(a)$ to $v_{next}(a)$ each of whose corresponding value of $Active_{now}$ is true. In Fig. 3 (b), if v_j and v'_j are active and v_ℓ and v'_ℓ are followers, a_i and other agents can get a_i's GID $GID_i = (2, 4)$.

Now, we explain implementation of the subphase. While follower agents in [12] stay at the current node, follower agents in this paper also move in the ring. First, each follower agent moves moves to the nearest active node to simulate the behavior of the active agent. To do this, each agent has variable $nearActive_{now}$ that indicates the number of tokens to the nearest active node



(a)



(b)

Figure 3: (a): A GID of an active agent a_i in [12]. (b): A GID of an active agent a_i in this section (v_j and v'_j are active and v_ℓ and v'_ℓ are followers).

in the subphase (the values of $nearActive_{now}$ for active agents are 0). Then, each active or follower agent a_i travels once around the ring. While traveling, a_i executes the following actions:

(1) Get its GID $GID_i = (fNum_i, d_i)$: Agent a_i gets its GID GID_i by moving from the current node to the next active node (i.e., from $v_{HOME}(a_i)$ to $v_{next}(a_i)$ for active agent a_i or from the nearest active node v to the next active node from v for follower agent a_i) with counting the numbers of followers and visited nodes (Fig. 3 (b)).

(2) Compare GID_i with GIDs of all active agents: During the traversal, a_i compares GID_i with GIDs of all active agents one by one, and checks 1) whether GID_i is the lexicographically maximum and 2) whether the GIDs of all active agents are the same. To check these, a_i keeps a variable *same* (*same* = *true* means that GIDs a_i ever found are the same). In addition, a_i keeps a variables GID_{max} that is the largest GID among GIDs a_i ever found, and a_i updates *same* and GID_{max} (if necessary) every time it finds a GID of another active agent. When GID_{max} is updated, a_i also updates the value of $nearActive_{next}$, indicating the number of tokens to the nearest active node in the next subphase.

When completing one circuit of the ring, a_i returns to $v_{HOME}(a_i)$ and determines its state for the next subphase. (a) If *same* = *true*, a_i (and all the other active agents) become leaders and completes the selection phase. (b) If *same* = *false* and $GID_i = GID_{max}$, a_i remains in its state (active or follower) and executes the next subphase. (c) If a_i does not satisfy (a) or (b), each active (resp., follower) agent becomes (resp., remains) a follower and executes the next subphase. By repeating such subphase at most $\lceil \log k \rceil$

times, all the remaining active agents end up with the same GID in some subphase and they are selected as leaders so that their home nodes should satisfy the base node conditions. Notice that $\lceil \log k \rceil$ subphases are sufficient, since no pair of adjacent active agents remain active in every subphase (as shown in the proof of Theorem 2).

The pseudocode of the selection phase is described in Algorithm 1. Variable t represents the number of tokens (mod k) that agent a_i has observed during traversal of the ring. Each agent uses variable *preActive* for storing the position (i.e., the ordinary number) of the last active node it visited before coming to the current node, and a k-bits boolean array Active_{next} for storing the states of all agents for the next subphase. In addition, agents use procedure *nextActive*() to move to the next active node, and its pseudocode is described in Procedure 1. Note that, in each subphase each follower agent firstly moves to the nearest active node, travels once around the ring from the active node, and returns to its home node. Hence, each follower agent travels twice around the ring in each subphase and each active agent does so for simplicity. In addition, in Algorithm 1 each agent can get the number n of nodes when it finishes traveling once around the ring, but we omit the description.

3.2.2. Deployment Phase

In this phase, each agent determines its destination node and moves to the node. If $v_{HOME}(a_i)$ is a base node (i.e., a_i is a leader), $v_{HOME}(a_i)$ is a_i's destination node and a_i stays there. Otherwise (i.e., if a_i is a follower), a_i firstly moves until it observes $\mathit{nearActive}_{now}$ tokens to reach the nearest base node. After this, a_i moves $\mathit{nearActive}_{now} \times n/k$ times to reach its destination node. For example, in Fig. 4 we assume that nodes v_d^0 and v_d^3 are selected as base nodes. Then, destination nodes of a_0, a_1, a_2, a_3, a_4, and a_5 are $v_d^0, v_d^1, v_d^2, v_d^3, v_d^4$, and v_d^5, respectively. When all agents move to their destination nodes, the final configuration is a solution of the uniform deployment problem.

The pseudocode of the deployment phase is described in Algorithm 2. We have the following theorem for the proposed algorithm.

Theorem 2. *For agents without multiplicity detection, the proposed algorithm solves the uniform deployment problem with termination detection. This algorithm requires $O(k + \log n)$ memory space per agent, $O(n \log k)$ time, and $O(kn \log k)$ total number of moves.*

Proof. At first, we show correctness of the algorithm. From lines 21 to 24 in

Algorithm 1 The behavior of active or follower agent a_i in the selection phase

Behavior of Agent a_i

```
1: /*selection phase*/
2: $phase = 1$, $t = 0$, $nearActive_{now} = 0$, $nearActive_{next} = 0$, $preActive = 0$, $same = true$
3: for $j = 0; j < k - 1; j++$ do $Active_{now}[j] = true$, $Active_{next}[j] = true$
4: release a token at its home node $v_{HOME}(a_i)$
5: while $phase \neq \lceil \log k \rceil$ do
6:   if $a_i$ is a follower then
7:     move until it observes $nearActive_{now}$ tokens // reach the nearest active node
8:     $t = nearActive_{now}$
9:   end if
10:  execute $NextActive()$ and get the first GID $GID_i = (fNum_i, d_i)$, $GID_{max} = GID_i$
11:  while $t \neq nearActive_{now}$ do
12:    execute $NextActive()$ and get GID $GID_{oth} = (fNum_{oth}, d_{oth})$ of the next active agent
13:    if $GID_{oth} \neq GID_i$ then $same = false$
14:    if $GID_{max} > GID_{oth}$ then $Active_{next}[preActive] = false$
15:    if $GID_{max} < GID_{oth}$ then
16:      $GID_{max} = GID_{oth}$, $nearActive_{next} = preActive$
17:      for $j = 0; j < t - 1; j++$ do $Active_{next}[j] = false$
18:    end if
19:  end while
20:  move until it observes $k - nearActive_{now}$ tokens // return to its home node $v_{HOME}(a_i)$
21:  if $same = true$ then // active nodes satisfy the base node conditions
22:    if $a_i$ is active then enter a leader state
23:    terminate the selection phase and enter the deployment phase
24:  end if
25:  if ($a_i$ is active) $\wedge$ ($GID_i \neq GID_{max}$) then enter a follower state
26:  $phase = phase+1$, $same = true$, $nearActive_{now} = nearActive_{next}$
27:  for $j = 0; j < k - 1; j++$ do $Active_{now}[j] = Active_{next}[j]$
28: end while
```

Procedure 1 Procedure *NextActive*()

Behavior of Agent a_i

```
1: $preActive = t$
2: move to the next token node and set $t = (t+1) \mod k$
3: **while** $Active_{now}[t] \neq true$ **do**
4:     move to the next token node and set $t = (t+1) \mod k$
5: **end while**
```

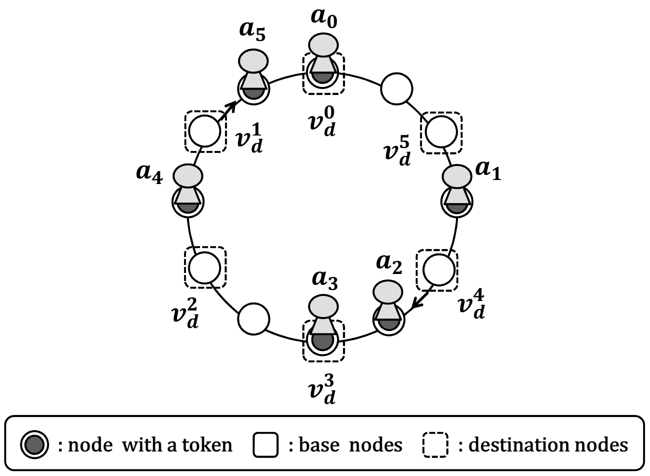


Figure 4: An example of the deployment phase.

Algorithm 1, when *same* = *true*, agents enter the deployment phase. This means that all the active nodes in the subphase have the same GID and they satisfy the base node conditions. In addition, from Algorithm 2 each agent can determine its destination node and moves to the node. Hence, the algorithm solves the uniform deployment problem. In the following, we analyze complexity.

At first, we evaluate the memory space per agent. Each agent a_i memorizes at most three GIDs GID_i, GID_{max}, and GID_{oth}, and each of them requires $O(\log n)$ memory space. In addition, a_i memorizes two boolean arrays $Active_{now}$ and $Active_{next}$, each of which requires $O(k)$ memory space. Since other variables require $O(k + \log n)$ memory space or less, each agent requires $O(k + \log n)$ memory space.

Next, we analyze the time complexity and the total number of moves. We show that base nodes are selected within at most $\lceil \log k \rceil$ subphases. To

Algorithm 2 The behavior of agent a_i in the deployment phase

```
Behavior of Agent $a_i$
 1: /*deployment phase*/
 2: if $a_i$ is a leader then
 3:     keep staying at the current node $v_{HOME}(a_i)$ and enter the halt state
        (or terminate the algorithm)
 4: end if
 5:
 6: if  $a_i$ is a follower then
 7:     move until it observes $nearActive_{now}$ tokens // reach the nearest base
        node
 8:     move $nearActive_{now} \times n/k$ times
 9:     enter the halt state
10: end if
```

show this, we first claim that the largest GID increases every time a subphase completes. That is, letting GID^j_{max} (resp., GID^{j+1}_{max}) be the maximum GID in the j-th (resp., $(j+1)$-st) subphase, then $GID^j_{max} < GID^{j+1}_{max}$ holds unless all the GIDs are the same in the j-th subphase. This property holds from the following reason. We assume that in the j-th subphase active agents $a^j_0, a^j_1, \ldots, a^j_{\ell-1}$ exist in this order. Since all the agents do not have the same GID in this subphase, at least one of the active agents becomes a follower. We assume that agent $a^j_{\ell'}$ $(0 \leq \ell' \leq \ell-1)$ becomes a follower and agent $a^j_{(\ell'-1) \bmod \ell}$ remains active in the j-th subphase. Then, $a^j_{(\ell'-1) \bmod \ell}$ clearly gets a GID larger than GID^j_{max} in the $(j+1)$-st subphase. Then, we show that base nodes are selected within $\lceil \log k \rceil$ subphases. Let A_j (resp., A_{j+1}) be a set of agents having GID^j_{max} (resp., GID^{j+1}_{max}) and remain active at the end of the j-th (resp., $(j+1)$-st) subphase. Then, $|A_{j+1}| \leq (1/2)|A_j|$ holds because 1) $GID^j_{max} < GID^{j+1}_{max}$ holds as mentioned above and 2) to satisfy $a_i \in A_j$ and $a_i \in A_{j+1}$ for some agent a_i, it is necessary that agent $a^j_{(i+1) \bmod \ell} \in A_j$ that is a_i's adjacent active agent in a_i's forward direction in the j-th subphase has to become a follower in the $(j+1)$-st subphase. Thus, base nodes are selected within at most $\lceil \log k \rceil$ subphases.

Now, we analyze the time complexity and the total number of moves. For each subphase in the selection phase, each agent travels twice around the ring, and each agent executes such a subphase at most $\lceil \log k \rceil$ times. Thus, the selection phase requires $O(n \log k)$ time units and $O(kn \log k)$ total number of moves. In the deployment phase, each agent moves to its destination node, which requires at most $2n$ time units and $2kn$ total number of moves. Hence, the algorithm requires the time complexity $O(n \log k)$ and the total number of moves $O(kn \log k)$. □

3.2.3. Uniform deployment for the case of $n \neq ck$

To remove the assumption of $n = ck$ imposed in Section 3, only the parts for determining the destination node and for moving to the destination node should be modified. In the case that n is not a multiple of k, the distance between adjacent destination nodes should be $\lceil n/k \rceil$ or $\lfloor n/k \rfloor$.

Since all the agents recognize a single base node or uniformly distributed base nodes, they can determine the uniformly distributed destination nodes using the base nodes as reference nodes: Let b be the number of the base nodes, and $r = n \bmod k$. The distance of every pair of adjacent base nodes is identical even in the case of $n \neq ck$, and is $n/b = (\lfloor n/k \rfloor \times k + r)/b = \lfloor n/k \rfloor \times k/b + r/b$ (notice that k/b and r/b are integers). This implies that we should select $k/b - 1$ destination nodes between two adjacent base nodes so that the first r/b intervals between adjacent destination nodes should be $\lceil n/k \rceil$ and others should be $\lfloor n/k \rfloor$. From the above discussion, we can see that each agent can determine its own destination node by local computation so that all the agents can spread over the ring to achieve uniform deployment.

4. Agents with weak multiplicity detection

In this section, we consider agents with weak multiplicity detection, and propose an algorithm to solve the uniform deployment problem without termination detection. This algorithm reduces the memory space per agent to $O(\log k + \log \log n)$, but uses $O(n^2 \log n)$ time and $O(kn^2 \log n)$ total number of moves. It also gives up the termination detection. The algorithm consists of three phases: the selection phase, the collection phase, and the deployment phase. In the selection phase, agents select base nodes similar to Section 3. In the collection phase, agents move in the ring so that they stay at consecutive nodes starting from the base nodes. In the deployment phase, agents move to their destination nodes. In this section, we assume that agents know an upper bound $\log N$ of $\log n$ such that $\log N = O(\log n)$.

4.1. Selection phase

Similar to Section 3, in this phase some home nodes are selected as base nodes. The basic idea is the same as that in Section 3, that is, agents find

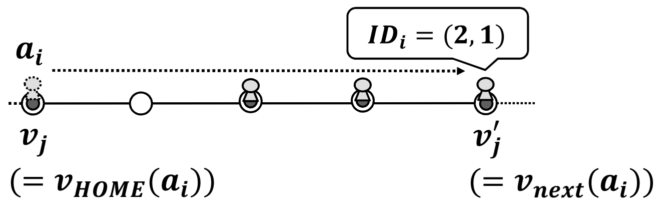


Figure 5: A GID of an active agent a_i ($prime_l = 3$).

GIDs and decrease the number of active agents using the GIDs. However, compared with the algorithm in Section 3, memory space is reduced from $O(k + \log n)$ to $O(\log k + \log \log n)$. We use two techniques for the reduction: (i) As in [12], a follower remains at its home node and informs an active agent of its state using the weak multiplicity detection: when an agent is detected at a node, it is recognized as a follower. This improves memory space from $O(k)$ to $O(\log k)$. The algorithm simply counts the number of followers between adjacent active nodes, while the algorithm in Section 3 requires $O(k)$ memory space to maintain the states of all agents. (ii) Distances are computed using Residue Number System (RNS) [16] that represents a large number as a set of small numbers. In particular, we use the technique called Chinese Remainder Theorem (CRT) [17]. The CRT says that for two positive integers n_1 and n_2 ($n_1, n_2 < n$), if the remainders when divided by each of the first $\log n$ prime numbers $2, 3, 5, \dots, U$ are the same, then $n_1 = n_2$ holds. The prime number theorem guarantees that the ($\log n$)-th prime U satisfies $U = O(\log^2 n)$. Thus, agents check if distances between adjacent active nodes are the same or not by using the CRT and reduce memory space from $O(\log n)$ to $O(\log \log n)$.

We explain the outline of the selection phase. As in Section 3, the state of an agent is active, leader, or follower, and initially all agents are active. At the beginning of the algorithm, each agent a_i releases its token at $v_{HOME}(a_i)$. The selection phase consists of at most $\lceil \log k \rceil$ subphases. As in Section 3, dropping out from active agents is realized by GIDs each of which consists of the number of followers and the distance between active nodes. The only difference is that the distance part is compared using remainders by primes (Fig. 5). Each subphase consists of several iterations. At the beginning of each iteration, each agent a_i stays at $v_{HOME}(a_i)$. For the l-th iteration in each subphase, if a_i is a follower, different from Section 3, it keeps staying $v_{HOME}(a_i)$ to inform active agents visiting the node of its state. On the other hand, each active agent a_i travels once around the ring and gets the distance part d_l^{prime} of its GID as the remainder divided by the l-th prime $prime_l$. In Fig. 5, when $prime_l = 3$, a_i gets its GID $GID_i = (2, 1)$.

During the traversal, a_i lexicographically compares its GID GID_i with GIDs of other active agents one by one, and it determines its next behavior when it returns to $v_{HOME}(a_i)$. As in Section 3, a_i uses variable *same* (*same* = *true* means that GIDs a_i ever found in the iteration are the same). Then, (a) if *same* = *true* and $l = \log N$, it means that the distances between all the pairs of adjacent active nodes are the same, and these home nodes satisfy the base node conditions. Hence, the active agents become leaders and enter the collection phase without staying at its home node. (b) If *same* = *true* but $l \neq \log N$, a_i executes the next $(l + 1)$-st iteration using the next prime $prime_{l+1}$. (c) If *same* = *false*, they terminate the current subphase. If a_i has the maximum GID, a_i remains active and starts the next subphase. Otherwise, a_i becomes a follower. Each active agent executes such subphases at most $\lceil \log k \rceil$ times. Notice that the distances are compared using the CRT, which implies that the agents with the maximum distance among the agents with the maximum *fNum* (the number of followers between adjacent active agents) do not necessarily remain active in the subphase. Hence, agents remaining active in the subphase may differ from those in the algorithm of Section 3. However, $\lceil \log k \rceil$ subphases are still sufficient as in Section 3, which is guaranteed by selecting active agents with the maximum *fNum*.

The pseudocode of the selection phase is described in Algorithm 3. Similar to Algorithm 1, variable t represents the number of tokens (mod k) that a_i observed during the travel of the ring. In addition, each agent a_i uses boolean variable *max* (*max* = *true* means GID_i is the maximum among GIDs a_i has ever found), and uses procedure *NextActive2*() to go the next active node and the pseudocode is described in Procedure 2.

4.2. Collection phase

In this phase, leader agents instruct follower agents so that they move to and stay at consecutive nodes starting from the base nodes. Concretely, each leader agent a_i firstly moves to the nearest follower node v_j (i.e., the token node with another agent) so that a_i makes the follower agent to execute the collection phase. Then, a_i waits at v_j until the follower leaves v_j. Note that, when an active agent in the selection phase visits v_j, it leaves v_j without staying there by the atomicity of an action. Hence, the behavior of leader agent a_i can inform a follower agent of the beginning of the collection phase.

Algorithm 3 The behavior of active agent a_i in the selection phase

```
Behavior of Active Agent $a_i$
1: /*selection phase*/
2: $phase = 1$, $prime = 2$, $t = 0$, $same = true$, $max = true$
3: release a token at its home node $v_{HOME}(a_i)$
4: while $(phase \neq \lceil \log k \rceil) \vee (prime \neq$ the $(\log N)$-th prime$)$ do
5:    execute NextActive2() and get its own GID $GID_i = (fNum_i, d_i^{prime})$
6:    while $t \neq 0$ do // $a_i$ is not at $v_{HOME}(a_i)$
7:       execute NextActive2() and get GID $GID_{oth} = (fNum_{oth}, d_{oth}^{prime})$ of the
            next active agent
8:       if $GID_{oth} \neq GID_i$ then $same = false$
9:       if $GID_{oth} > GID_i$ then $max = false$ // there exists an agent having
      a   larger GID
10:   end while
11:   // $a_i$ returns to its home node $v_{HOME}(a_i)$
12:   if $(same = true) \wedge (prime =$ the $(\log N)$-th prime$)$ then terminate
      the selection phase, start the collection phase with a leader state, and
      leave the current node // all active agents have the same GID for all
      target primes
13:   else if $(same = true) \wedge (prime \neq$ the $(\log N)$-th prime$)$ then $prime =$
      (next prime)
14:   else if $max = false$ then terminate the selection phase and start the
      collection phase with a follower state
15:   else $phase = phase + 1$, $prime = 2$, $same = true$, $max = true$
16: end while
```

Procedure 2 Procedure *NextActive2*()

```
Behavior of Active Agent $a_i$
1: move to the next token node and set $t = (t + 1) \mod k$
2: while another agent does not exist at the current node do
3:    move to the next token node and set $t = (t + 1) \mod k$
4: end while
```

After this, a_i leaves v_j and moves to the next follower node. This process is repeated until a_i reaches the next leader node (i.e., the token node with no agent). Note that, by the atomicity of an action and the FIFO property, when an agent moves to some leader node, the leader agent was already

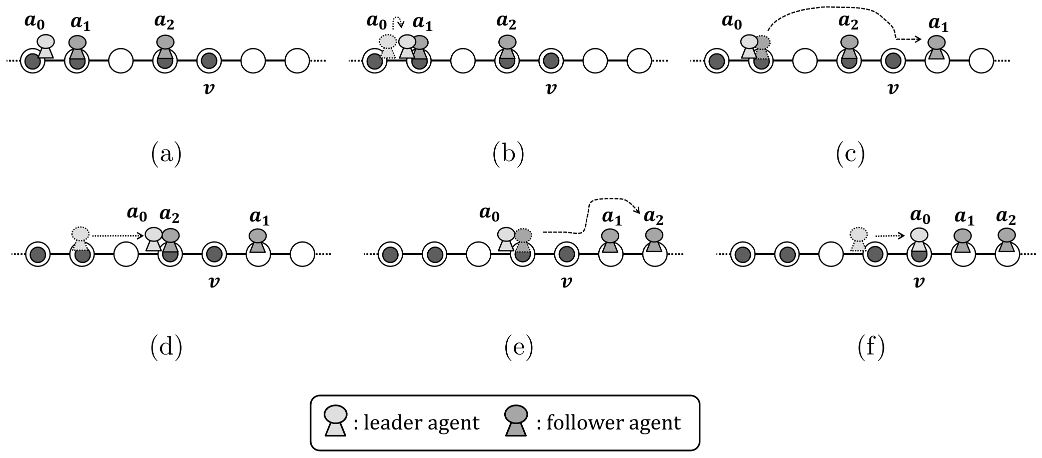


Figure 6: An example of the collection phase ($fNum = 2$).

activated, started its collection phase, and left the leader node. Hence, when agent a_i visits a token node with no agent, it can recognize that the nodes is the next leader node. On the other hand, each follower agent a_i waits at the current node until another agent (i.e., a leader) comes. Then, a_i firstly moves to the nearest leader node. After this, a_i moves until it reaches a node with no agent and stays there. When all agents finish their movements, the agents are divided into groups (possibly only one group) each of which consists of $fNum + 1$ agents, where $fNum$ is the number of follower agents between two adjacent leader agents when the selection phase completes, and the agents in a group are deployed at consecutive nodes starting from a base node.

For example, in Fig. 6 there exist one leader agent a_0 and two follower agents a_1 and a_2 between a_0 and its next leader (i.e., $fNum = 2$). From (a) to (b), a_0 firstly moves to the nearest token node with an agent (i.e., follower node), and stays there until the follower agent leaves the node. From (b) to (c), a_1 detecting another agent a_0 firstly moves to the token node with no agent (i.e., leader node v), and then moves to the next node. From (c) to (d), a_0 similarly moves to the next follower node where agent a_2 exists. From (d) to (e), a_2 firstly moves to leader node v and moves until it visits a node with no agent. From (e) to (f), a_0 moves to leader node v and finishes the collection phase. The pseudocode of the collection phase is described in Algorithm 4.

Algorithm 4 The behavior of leader or follower agent a_i in the collection phase (v_j is the current node of a_i)

```
Behavior of Agent $a_i$
1:  /*collection phase*/
2:  // the behavior of leader agents
3:  if $a_i$ is in the leader state then
4:      move to the next token node
5:      while there exists another agent at $v_j$ do
6:          wait at $v_j$ until there exists no agent other than $a_i$
7:          move to the next token node
8:      end while
9:      terminate the collection phase, start the deployment phase, and leave
        the current node
10: end if
11:
12: // the behavior of follower agents
13: if $a_i$ is in the follower state then
14:     wait at $v_j$ until there exists another agent
15:     move to the token node with no agent
16:     move to the next node
17:     while there exists another agent at $v_j$ do
18:         move to the next node
19:     end while
20:     terminate the collection phase and start the deployment phase
21: end if
```

4.3. Deployment phase

In this phase, leader agents control follower agents so that they should move to and stay at their destination nodes to achieve uniform deployment. Since each agent only has $O(\log k + \log\log n)$ memory space, it cannot measure explicitly the distance to its destination node and move to the node as in Section 3. Agents overcome such a problem using the weak multiplicity detection and the CRT. The basic idea is as follows. The deployment phase consists of several subphases, and the distance between every pair of adjacent agents in the same group is increased by one in each subphase. To realize it, each subphase consists of several iterations. For explanation of an iteration, consider a group where a_0 is a leader and followers $a_1, a_2, \ldots, a_{fNum}$

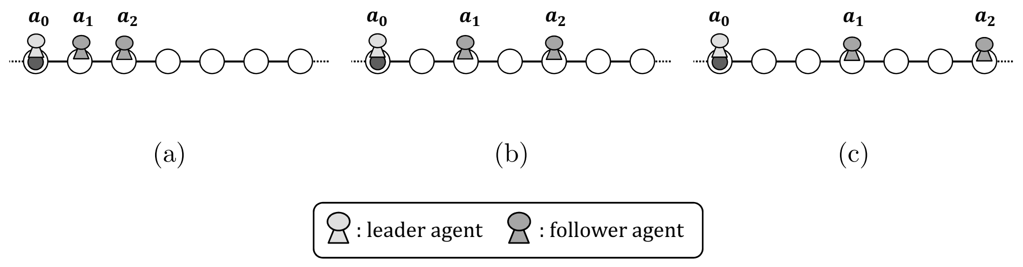


Figure 7: Behavior outline of the deployment phase for the case of $\mathit{fNum} = 2$ ((a): beginning of the phase, (b): the end of the first subphase, (c): the end of the second subphase)

are following a_0 in this order. At the beginning of the first subphase, they stay at consecutive nodes, and at the end of the h-th subphase the distance between every pair of adjacent agents becomes $h + 1$ (Fig. 7). Each subphase consists of *fNum* iterations. In the l-th iteration, each of the l agents $a_{\mathit{fNum}-l+1}, a_{\mathit{fNum}-l+2}, \ldots, a_{\mathit{fNum}}$ moves to the next node. Consequently, in each subphase a_m moves m times and thus the distance between every pair of adjacent agents increases by one.

The l-th iteration is realized as follows. Leader agent a_0 firstly moves to the node where $a_{\mathit{fNum}-l+1}$ is staying and stays there until $a_{\mathit{fNum}-l+1}$ moves to the next node. Then, a_0 moves to the node where $a_{\mathit{fNum}-l+2}$ is staying to make $a_{\mathit{fNum}-l+2}$ to move to the next node. This process is repeated until a_{fNum} moves to the next node. After this, a_0 makes a remaining circuit of the ring, returns to the node where it started the deployment phase, say $v_{dep}(a_0)$, and terminates the l-th iteration. Then, a_0 checks if the locations of agents from $v_{dep}(a_0)$ to the next leader node are uniform or not using the CRT. If the locations are uniform, a_0 returns to $v_{dep}(a_0)$ and enters a suspended state. Otherwise, a_0 executes the next iteration. When a_0 executes the *fNum*-th iteration and the locations are not uniform, a_0 executes the next subphase.

For example, in Fig. 8 there exist one leader agent a_0 and two follower agents a_1 and a_2 (i.e., *fNum*=2). Variable v' represents the next leader node. Let d_1 (resp., d_2) be the distance from a_0 to a_1 (resp., a_1 to a_2), and d_3 be the distance from a_2 to v'. In (a), $d_1 = d_2 = 1$ and $d_3 = 4$ holds. From (a) to (b), as the first iteration in the first subphase, a_0 moves to the node where the *fNum*-th follower agent (i.e., a_2) exists and stays there until a_2 moves to the next node. From (b) to (c), a_0 returns to the node $v_{dep}(a_0)$ where it started the deployment phase. Then, $d_1 = 1, d_2 = 2$, and $d_3 = 3$ hold. Since

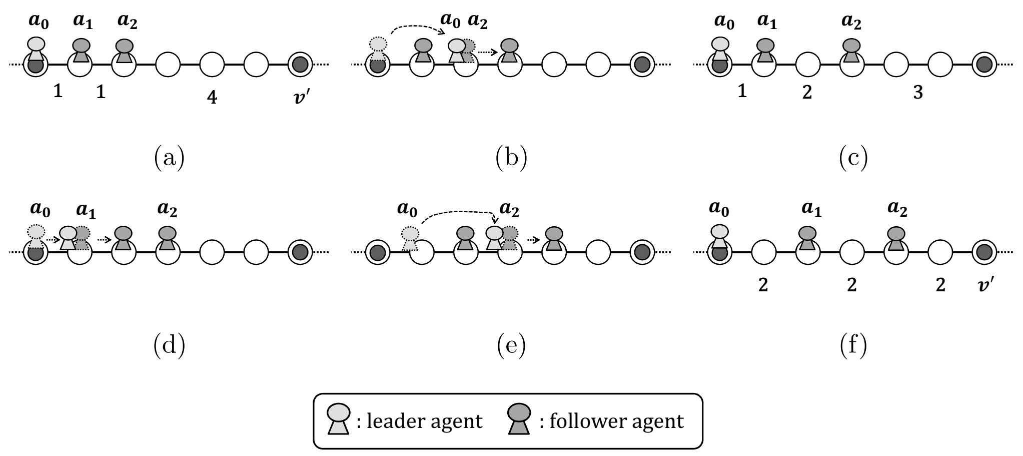


Figure 8: An example of the deployment phase (*fNum* $= 2$, v' is the next leader node).

a_0 recognizes that the locations of agents are not uniform, it executes the next iteration. From (c) to (d), as the second iteration in the first subphase, a_0 moves to the node where the (*fNum*-1)-st agent (i.e., a_1) exists and stays there until a_1 moves to the next node. From (d) to (e), a_0 moves to the next follower's (i.e., a_2's) node and stays there until a_2 moves to the next node. From (e) to (f), a_0 returns to node $v_{dep}(a_0)$. Then, since $d_1 = d_2 = d_3 = 2$ holds, a_i recognizes that the locations of agents becomes uniform.

The pseudocode of the deployment phase is described in Algorithm 5. In Algorithm 5, each leader agent uses procedure *Check*() to check whether the locations of agents are uniform or not by comparing the distance dis_1 from the (*fNum*-1)-th follower agent to the *fNum*-th follower agent with the distance dis_2 from the *fNum*-th follower agent to the next leader node using the CRT. In *Check*(), variable *diff* represents the difference between dis_1 and dis_2 (i.e., $diff = dis_1 - dis_2$), and the pseudocode is described in Procedure 3. We have the following theorem for the proposed algorithm.

Theorem 3. *For agents with weak multiplicity detection and knowledge of an upper bound* $\log N$ *of* $\log n$ *satisfying* $\log N = O(\log n)$*, the proposed algorithm solves the uniform deployment problem without termination detection. This algorithm requires* $O(\log k + \log\log n)$ *memory space per agent,* $O(n^2 \log n)$ *time, and* $O(kn^2 \log n)$ *total number of moves.*

Proof. At first, we show correctness of the proposed algorithm. From line

Algorithm 5 The behavior of leader or follower agent a_i in the deployment phase (v_j is the current node of a_i)

```
Behavior of Agent $a_i$
 1: /*deployment phase*/
 2: // the behavior of leader agents
 3: if $a_i$ is in the leader state then
 4:    while true do
 5:       for $l = 1$ to $fNum_i$ do
 6:          for $m = l$ to 1 do
 7:             move to the node where the $(fNum + 1 - m)$-th follower agent
                exists
 8:             wait at $v_j$ until there exists no agent other than $a_i$
 9:          end for
10:          return to the node where it started the deployment phase
11:          $Check()$
12:       end for
13:    end while
14: end if
15:
16: // the behavior of follower agents
17: if $a_i$ is in the follower state then
18:    enter a suspended state
19:    while true do
20:       wait at $v_j$ until there exists another agent
21:       move to the next node and enter a suspended state
22:    end while
23: end if
```

12 of Algorithm 3, at the end of the selection phase, the active nodes (or the home nodes of the selected leaders) satisfy the base node conditions. In addition, from Algorithm 4, at the end of the collection phase agents are divided into groups each of which consists of $fNum+1$ agents and agents in a group are staying at the consecutive nodes starting at a base node. Without loss of generality, in the following we consider a group of agents between two adjacent base nodes v_{base} (inclusively) and v'_{base} (exclusively). We denote by d_{base} the distance from v_{base} to v'_{base}. Then, at the end of the collection phase the distance sequence D_{group} of the group of agents can be represented by

Procedure 3 Procedure *Check*()

Behavior of Agent a_i

```
1: $\mathit{diff} = 0$, $\mathit{uniform} = \mathit{true}$
2: for $\mathit{prime} = 2, 3, 5, \dots,$ the $(\log N)$-th prime do
3:   move to the $(\mathit{fNum}_i - 1)$-th node $v$ among nodes where an agent exists with counting the number $t$ of tokens // $v$ is the node where the $(\mathit{fNum}_i - 1)$-st follower is staying
4:   move to the next node $v'$ where an agent exists // $v'$ is the node where $(\mathit{fNum}_i)$-th follower is staying
5:   $\mathit{dis}_1 =$ (distance from $v$ to $v'$) mod $\mathit{prime}$
6:   move until it observes $(\mathit{fNum} - t)$ tokens // reach the next base node
7:   let $v_{\mathit{nBase}}$ is the current node
8:   $\mathit{dis}_2 =$ (distance from $v'$ to $v_{\mathit{nBase}}$) mod $\mathit{prime}$
9:   return to the node where it started the deployment phase
10:  if $\mathit{prime} = 2$ then
11:    if $(\mathit{dis}_1 \neq \mathit{dis}_2) \vee (\mathit{dis}_1 \neq \mathit{dis}_2 + 1)$ then $\mathit{uniform} = \mathit{false}$ and break // the locations are not uniform
12:    else $\mathit{diff} = \mathit{dis}_1 - \mathit{dis}_2$
13:  else // $\mathit{prime} \geq 3$
14:    if $(\mathit{dis}_1 - \mathit{dis}_2 \neq \mathit{diff})$ then $\mathit{uniform} = \mathit{false}$ and break // the locations are not uniform
15: end for
16: if $\mathit{uniform} = \mathit{true}$ then
17:   // locations of agents from the node where it started the deployment phase to the next leader node is uniform
18:   enter a suspended state
19: end if
```

$(1, 1, \dots, 1, d_{base} - \mathit{fNum})$.

Now, we consider the deployment phase. We first claim that at the beginning of the d'-th subphase (for every d') the distance sequence D_{group} can be represented by $D_{group} = (d', d', \dots, d', d_{base} - (d' - 1) \times \mathit{fNum})$, and at the end of the l-th iteration in the d'-th subphase D_{group} can be represented by $D_{group} = (\underbrace{d', d', \dots, d'}_{\mathit{fNum}\text{-}l}, \underbrace{d' + 1, d' + 1, \dots, d' + 1}_{l}, d_{base} - (d' - 1) \times \mathit{fNum} - l)$. This is because, from lines 5 to 12 and 17 to 23 in Algorithm 5, during the first l iterations in each subphase the $(\mathit{fNum} + 1 - m)$-th follower agent

$(1 \leq m \leq l)$ moves $l - m + 1$ times. Hence, since the *fNum*-th follower moves exactly once in each iteration and thus moves *fNum* times in each subphase. Precisely, the m-th follower agent moves exactly m times in each subphase, which leads to the above claims. Let $d'_{x(l)} = d_{base} - (d' - 1) \times \mathit{fNum} - l$. Then, since the value of $d'_{x(l)}$ decreases one by one during execution of the deployment phase, there exists some l-th iteration in the d'-th phase such that $d'_{x(l)} = d'$ or $d'_{x(l)} = d' + 1$ holds. Then, at the end of the iteration the locations of agents are uniform. This condition is checked by each leader agent using Procedure *Check*() to compare the distance from the $(\mathit{fNum} - 1)$-th follower agent to the *fNum*-th follower agent (i.e., $d' + 1$) with the distance from the *fNum*-th follower agent to v'_{base} (i.e., $d'_{x(l)}$). Thus, each leader agent can detect that the locations of agents are uniform, and after this it enters a suspended state at the node where it started the deployment phase. Thus, the algorithm can solve the uniform deployment problem.

In the following, we analyze complexity. At first, we evaluate the memory space per agent. Each agent a_i has two variables GID_i and GID_{oth} to store GIDs, each of which requires $O(\log k + \log\log n)$ memory space. Since other variables require $O(\log k + \log\log n)$ memory space or less, each agent requires $O(\log k + \log\log n)$ memory space.

Next, we analyze the time complexity and the total number of moves. For each subphase in the selection phase, each active agent travels at most $O(\log n)$ times to check whether home nodes of active agents satisfy the base node conditions or not using the Chinese Remainder Theorem. Since each active agent executes such a subphase at most $\lceil \log k \rceil$ times and there are at most k active agents, the selection phase requires $O(n \log k \log n)$ time units and $O(kn \log k \log n)$ total number of moves. Next, in the collection phase each leader agent moves to the next leader node, and each follower agent moves to the nearest leader node and then moves to the nearest empty (i.e., no agents exist) node. Hence, each of movements for leader agents and follower agents requires $O(n)$ time units and $O(kn)$ total number of moves. Finally, we consider the deployment phase. For each iteration in each subphase, each leader agent travels once around the ring to make follower agents move, and travels at most $O(\log n)$ times around the ring to check whether the locations of agents are uniform or not using the Chinese Remainder Theorem. Since one subphase consists of at most k iterations and there exist at most k leader agents in each subphase, agents require $k \times (n + n \log n) = O(kn \log n)$ time units and $O(k^2 n \log n)$ total number of moves to execute one subphase. Moreover, since agents execute such a subphase at most $\lceil n/k \rceil$ times, the deployment phase requires $O(n^2 \log n)$ time units and $O(kn^2 \log n)$ total number of moves. Therefore, the proposed algorithm requires $O(n^2 \log n)$ time units and $O(kn^2 \log n)$ total number of moves. $\square$

5. Conclusion

In this paper, we proposed two space-efficient uniform deployment algorithms in asynchronous unidirectional ring networks. For agents without multiplicity detection, we showed that each agent requires $\Omega(\log n)$ memory space, and proposed an algorithm to solve the problem with termination detection that requires $O(k + \log n)$ memory space per agent, $O(n \log k)$ time, and $O(kn \log k)$ total number of moves. This algorithm is optimal in memory space per agent when $k = O(\log n)$. For agents with weak multiplicity detection, we proposed an algorithm to solve the problem without termination detection that requires $O(\log k + \log \log n)$ memory space per agent, $O(n^2 \log n)$ time, and $O(kn^2 \log n)$ total number of moves.

Open problems are as follows. The first is concerning agents without multiplicity detection and to propose a space-optimal (i.e., $O(\log n)$ memory) algorithm to solve the problem. The second is concerning agents with weak multiplicity detection and to show a lower bound of memory space per agent. We conjecture that it is $\Omega(\log k + \log \log n)$, which implies that the second algorithm is asymptotically optimal in memory space per agent. The third is to analyze the space complexity with relaxed models, e.g., a model such that each agent can have more than one tokens and/or depends on randomness. The last one is to consider solvability in bidirectional rings. In particular, when agents do not have sense of direction and the initial deployment of agents is symmetric, it is possible that agents cannot achieve uniform deployment.

References

[1] S. R. Gray, D. Kotz, G. Cybenko, and D Rus. D'agents: Applications and performance of a mobile-agent system. Softw., Pract. Exper., 32(6):543–573, 2002.

[2] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. Mole–concepts of a mobile agent system. World Wide Web, 1(3):123–137, 1998.

[3] D.B. Lange and M. Oshima. Seven good reasons for mobile agents. Commun. ACM, 42(3):88–89, 1999.

[4] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agent coordination for distributed network management. Journal of Network and Systems Management, 9(4):435–456, 2001.

[5] A. Bieszczad, B. Pagurek, and T. White. Mobile agents for network management. IEEE Communications Surveys, 1(1):2–9, 1998.

[6] E. Kranakis and D. Krizanc. An algorithmic theory of mobile agents. International Symposium on Trustworthy Global Computing, Vol. 4661. pages 86-97, 2006.

[7] S. Lipperts and B. Kreller. Mobile agents in telecommunications networks - a simulative approach to load balancing. Proc. of 5th Information systems analysis an d synthesis, pages 231–238, 1999.

[8] J. Cao, Y. Sun, X. Wang, and S.K. Das. Scalable load balancing on distributed web servers using mobile agents. Journal of Parallel and Distributed Computing, 63(10):996–1005, 2003.

[9] P. Flocchini, G. Prencipe, and N. Santoro. Self-deployment of mobile sensors on a ring. Theoretical Computer Science, 402(1):67–80, 2008.

[10] E. Yotam and B. M. Alfred. Uniform multi-agent deployment on a ring. Theoretical Computer Science, 412(8):783–795, 2011.

[11] L. Barriere, P. Flocchini, E. Mesa-Barrameda, and N Santoro. Uniform scattering of autonomous mobile robots in a grid. International Journal of Foundations of Computer Science, 22(03):679–697, 2011.

[12] M. Shibata, T. Mega, F. Ooshita, H. Kakugawa, and T. Masuzawa. Uniform deployment of mobile agents in asynchronous rings. Proc. of the ACM Symposium on Principles of Distributed Computing, pages 415–424, 2016.

[13] AD. Kshemkalyani and F. Ali. Efficient dispersion of mobile robots on graphs. Proc. of the 20th International Conference on Distributed Computing and Networking, pages 218–227, 2019.

[14] A. Agarwalla, J. Augustine, WK. Moses Jr, SK. Madhav, and AK. Sridhar. Deterministic dispersion of mobile robots in dynamic rings. Proc. of the 19th International Conference on Distributed Computing and Networking, page 19, 2018.

[15] G. Tel. Introduction to distributed algorithms. Cambridge university press, 2000.

[16] OR. Amos and P. Benjamin. Residue number systems: theory and implementation, volume 2. World Scientific, 2007.

[17] D. Pei, A. Salomaa, and C. Ding. Chinese remainder theorem: applications in computing, coding, cryptography. World Scientific, 1996.