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

Masahiro Shibata[†], Hirotsugu Kakugawa[◇], and Toshimitsu Masuzawa[◇]

[†]Department of Computer Science and Electronics, Kyushu Institute of Technology
680-4, Kawadu, Iizuka, Fukuoka, 820-8502, Japan
Email:`shibata@cse.kyutech.ac.jp`, Tel.:+81 9 4829 7656
[◇]Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871, Japan
Email:`{kakugawa, masuzawa}@ist.osaka-u.ac.jp`, Tel.:+81 6 6879 4117

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 another agent staying at the same node, but cannot learn the exact number. Then, we show that the memory space per agent can be reduced to $O(\log k + \log \log n)$. 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.

keyword: 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]. Agents traverse the system carrying information collected at visited nodes and process tasks on each node using the information. In other words, agents encapsulate the process code and data, which simplifies design of distributed systems [2].

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

[⋆] A preliminary brief announcement of this work appeared in the proceedings of the 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2017). This work was partially supported by JSPS KAKENHI Grant Number 16K00018, 17K19977, and 18K18031, and Japan Science and Technology Agency (JST) SICORP.

uniformly in the network. Uniform deployment is useful for network management. In a distributed system, it is necessary that regularly each node gets software updates and is checked whether some application installed on the node is running correctly or not [3]. 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 [4]. 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, the uniform deployment problem is considered in ring networks [5, 6] and grid networks [7]. All of them assumed that agents are oblivious (or memoryless) but can observe multiple nodes within its visibility range. On the other hand, our previous work [8] considered uniform deployment in asynchronous unidirectional ring networks for agents that have memory but cannot observe nodes except for their currently visiting nodes.

1.2 Our contribution

In this paper, we consider the uniform deployment problem in asynchronous unidirectional ring networks. Similarly to [8], we consider agents that have memory but cannot observe nodes except for their currently visiting nodes. While the previous work [8] 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 [8]. That is, while agents in [8] assumed that they 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. We assume that agents have knowledge of the number k of agents.

In Table 1, we compare our contributions with the results for agents with knowledge of k in [8]. We consider two problem settings. The first setting considers agents *without* multiplicity detection, that is, agents cannot detect whether another agent 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 $O(k + \log n)$ memory space per agent, $O(n \log k)$ time, and $O(kn \log k)$ total number of moves. The second setting considers agents *with* the weak multiplicity detection, that is, agents can detect another agent staying at the same node, but cannot learn the exact number. 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 reduce the memory space per agent to $O(\log k + \log \log n)$, but it 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

Table 1. Results for agents with knowledge of k (n, #nodes, k, #agents)

	Previous results [8]		Results of this paper	
	Result 1	Result 2	Model 1	Model 2
Communication	Messages	Messages	Unremovable tokens	Unremovable tokens
Multiplicity detection	Required	Required	Not Required	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	$\Theta(kn)$	$\Theta(kn)$	$O(kn \log k)$	$O(kn^2 \log n)$

research considering the effect of the multiplicity detection on memory space required to solve problems.

Due to limitation of space, we omit several pseudocodes and proofs of theorems.

2 Preliminaries

2.1 System model

We use almost the same model as that in [8]. 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 $E = \{e_0, e_1, \ldots, e_{n-1}\}\,(e_i = (v_i, v_{(i+1) \bmod n}))$. We define the direction from v_i to v_{i+1} as the *forward* direction. In addition, we define the i-th ($i \neq 0$) (forward) agent a' of agent a as the agent such that $i-1$ agents exist between a and a' in a's forward direction. Moreover, 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. Each agent initially has a *token* and can release it on a visited node. 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[1]. 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 cannot learn the exact

[1] In practice, each node can store more information, but it is sufficient to store information about tokens when considering anonymous agents.

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

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

number[2]. 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 $S = (s_0, s_1, \dots, s_{k-1})$, 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 $T = (t_0, t_1, \dots, t_{n-1})$, 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 $P = (p_0, p_1, \dots, p_{n-1})$, where p_i is a set of agents staying at node v_i. Element Q is an n-tuple $Q = (q_0, q_1, \dots, q_{n-1})$, 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[3], and no node has any token. 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 $v_{HOME}(a)$.

A (sequential) *schedule* $X = \rho_0, \rho_1, \dots$ 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 $E = C_0, C_1, \dots$ is called an *execution* from C_0 if there exists a fair schedule $X = \rho_0, \rho_1, \dots$ that satisfies the following conditions for each h ($h > 0$):

[2] This is why such multiplicity detection is called *weak*.

[3] We assume this 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.

– If agent $\rho_{h-1} \in p_i$ (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, 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. 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 *without termination detection*. In this case, *suspended states* are defined as follows. An agent stays at a node (not in a link) when it is at 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. The uniform deployment problem without termination detection allows agents to stop in suspended states, which is also known as communication deadlock. We define the problem as follows.

Definition 1. *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, $q_i = \emptyset$ holds for any $q_i \in Q$ and the distance of each pair of adjacent agents is $\lfloor n/k \rfloor$ 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 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)[4]. 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. *For agents without multiplicity detection, the memory space per agent to solve the uniform deployment problem is $\Omega(\log n)$.* □

Proof. We show the theorem by contradiction. We assume that there exists an algorithm to solve the uniform deployment problem with at most $\log n - 2$ bit memory per agent. Then, each agent has at most $2^{(\log n - 2)} = n/4$ states. Hence, when an agent enters a suspended state, it moved at most $n/4$ times after it last observed a token.

We consider the initial configuration such that two agents a_1 and a_2 are placed at neighboring nodes in a n-node ring. Then, the distance between the two agents in the final configuration should be $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$. We assume that a_1 and a_2 move in a synchronous manner. Then, since they are placed at neighboring nodes and execute the same algorithm, they release tokens (if do) also at neighboring nodes. In addition, since a_1 and a_2 move at most $n/4$ times after they last observed a token and enter suspended states, the distance between them is at most $n/4+1 (\neq \lfloor n/2 \rfloor$ or $\lceil n/2 \rceil)$. However, this contradicts the condition of uniform deployment. □

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 $O(k + \log n)$ 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 [8]: 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 *target node* where it should enters a suspended state and moves to the node. For simplicity, we assume $n = ck$ for some positive integer c since we can easily remove this assumption, but we omit the description.

[4] This definition is based on the ideal time complexity for asynchronous message-passing systems [9].

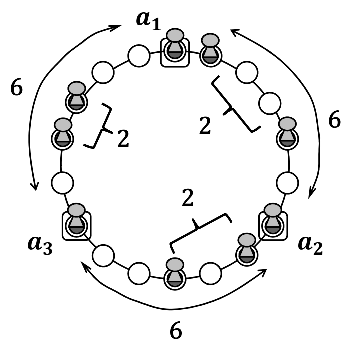
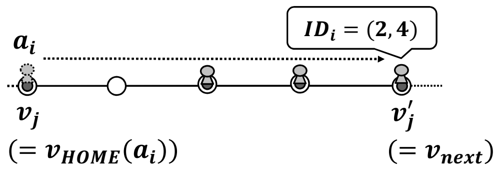


Fig. 1. An example of the base node conditions ($n = 18, k = 9$).

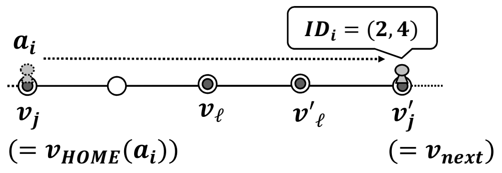
3.2.1 Selection Phase In this phase, some home nodes are selected as base nodes. The selected base nodes satisfy the following conditions called the **base node conditions**: 1) At least one base node exists, 2) the distance between every pair of *adjacent base nodes* is the same, and 3) the number of home nodes between every pair of adjacent base nodes is the same. We say that two base nodes are adjacent when there exists no base node between them. In Fig. 1, 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. 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 probably not unique) when its home node is selected as a base node, and call it a *follower* otherwise. The state of an agent is active, leader or follower. Active agents are candidates for leaders, and initially all agents are active. We say that node v is active (resp., follower) when v is the home node of an active (resp., a follower) agent.

At first, we explain the basic idea of the selection phase in [8], which assumes weak multiplicity detection, and then we explain the way of applying the idea to the model in this section. In the selection phase of [8], agents use *IDs* (but probably not unique) and decrease the number of active agents. We explain the detail of the IDs later. 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 ID by the method described later[5]. Then, a_i compares its ID with IDs of other agents one by one (a_i gets them during the traversal of the ring) and determines the next behavior. Briefly, (a) if all active agents have the same ID, 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 ID but a_i's ID is the maximum, it remains active

[5] Each agent can detect when it completes one circuit of the ring using knowledge of k.



(a)



(b)

Fig. 2. (a): An ID of an active agent a_i in [8]. (b): An ID of an active agent a_i in this section (v_j and v'_j are active and v_ℓ and v'_ℓ are followers).

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 ID. The ID (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}, and d_i is the distance from $v_{HOME}(a_i)$ to v_{next}. In Fig. 2 (a), when agent a_i moves from its home node $v_j(= v_{HOME}(a_i))$ to the next active node $v'_j(= v_{next})$, it observes two follower nodes and visits four nodes. Hence, a_i gets its ID $ID_i = (2, 4)$. Note that since active agents traverse the ring and follower agents stay at their home nodes, a_i can detect its arrival at the next active node when it visits a token node with no agent. This statement holds even in asynchronous systems by the FIFO property of links and the atomicity of execution (these facts are used in Section 4). Agents in [8] use $O(\log n)$ memory space to get such an ID and decide whether they remain active (or they have the lexicographically maximum ID) or not. Notice that an agent may get different IDs in different subphases.

In the following, we explain how to apply the previous idea to the model in this section (i.e., 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, each agent memorizes the state of all agents by using an array Active_{now} of k bits. The value of $\mathit{Active}_{now}[i]$ is *true* iff its i-th agent is active (otherwise it is a follower). Hence, agents can get an ID by going from node v to v' each of whose corresponding value of Active_{now} is true. In Fig. 2 (b), if v_j and v'_j are active and v_ℓ and v'_ℓ are followers, a_i can gets its ID $ID_i = (2, 4)$. In addition, each follower agent also moves in the ring instead of staying at its home node, and we explain this next.

Now, we explain implementation of the subphase. Each follower agent firstly moves to the nearest active node to simulate the behavior of the active agent. To do this, each agent has variable $\mathit{nearActive}_{now}$ that indicates the number of tokens to the nearest active node in the subphase (the values of $\mathit{nearActive}_{now}$ 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 ID $ID_i = (\mathit{fNum}_i, d_i)$: Agent a_i gets its ID ID_i by moving from the current node (i.e., $v_{HOME}(a_i)$ for active agent a_i or the nearest active node for follower agent a_i) to

v_{next} with counting the numbers of followers and visited nodes (Fig. 2 (b)).

(2) Compare ID_i with IDs of all active agents: During the traversal, a_i compares ID_i with IDs of all active agents one by one, and checks 1) whether ID_i is the lexicographically maximum and 2) whether the IDs of all active agents are the same. To check these, a_i keeps variables ID_{max} that is the largest ID among IDs a_i ever found, and *same* (*same* $=$ *true* means that IDs a_i ever found are the same), and it updates the variables (if necessary) every time it finds an ID of another active agent. When ID_{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 $ID_i = ID_{max}$, a_i remains 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 become to have the same ID 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, intuitively because 1) the largest ID increases every time a subphase completes, and thus 2) no pair of adjacent active agents remain active in every subphase.

Pseudocode is described in Algorithm 1. Each agent uses variable *preActive* for storing the position (i.e., the ordinary number) of the active node it visited for the last time before coming to the current node, and boolean array $Active_{next}$ of k bits for storing the states of all agents for the next subphase. In addition, agents use procedure *nextActive*() to move to the next active node. 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 target node and moves to the node. At first, the nearest base node is first selected as the base node. Hence, if $v_{HOME}(a_i)$ is a base node (i.e., a_i is a leader), $v_{HOME}(a_i)$ is a_i's target node and a_i stays there. Otherwise (i.e., if a_i is a follower), a_i firstly moves until it observes $nearActive_{now}$ tokens to reach the nearest base node. After this, a_i moves $nearActive_{now} \times n/k$ times to reach its target node. When all agents move to their target nodes, the final configuration is a solution of the uniform deployment problem.

We have the following theorem for the proposed algorithm.

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

```
Behavior of Agent $a_i$
1: /*selection phase*/
2: $phase = 1$, $nearActive_{now} = 0$, $nearActive_{next} = 0$, $preActive = 0$, $same = true$
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 ID $ID_i = (fNum_i, d_i)$, $ID_{max} = ID_i$
11:   while $t \neq nearActive_{now}$ do
12:      execute $NextActive()$ and get ID $ID_{oth} = (fNum_{oth}, d_{oth})$ of the next active agent
13:      if $ID_{oth} \neq ID_i$ then $same = false$
14:      if $ID_{max} > ID_{oth}$ then $Active_{next}[preActive] = false$
15:      if $ID_{max} < ID_{oth}$ then
16:         $ID_{max} = ID_{oth}$, $nearActive_{next} = preActive$
17:         for $j = 0; j < t-1; j++$ do $Active_{next}[j] = false$
18:      end if
19:   end while
20:   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) $\wedge$ ($ID_i \neq ID_{max}$) 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
29:
Procedure NextActive()
30: $preActive = t$
31: move to the next token node and set $t = (t+1) \mod k$
32: while $Active_{now}[t] \neq true$ do
33:    move to the next token node and set $t = (t+1) \mod k$
34: end while
```

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

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 that reduces the memory space per agent to $O(\log k + \log \log n)$, but it uses $O(n^2 \log n)$ time and $O(kn^2 \log n)$ total number of moves. The algorithm consists of three phases: the selection phase, the collection phase, and the deployment phase. In the selection phase, agents select base nodes similarly 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 target nodes. In this section, we assume that agents know an upper bound $\log N$ of $\log n$ such that $\log N = O(\log n)$.

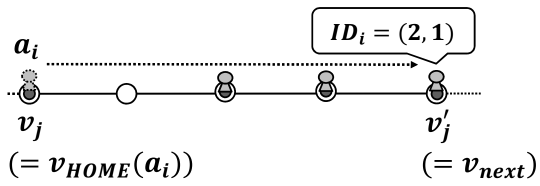


Fig. 3. An ID of an active agent a_i ($prime_l = 3$).

4.1 Selection phase

Similarly 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 use IDs and decrease the number of active agents. However, compared with the algorithm in Section 3, memory space is reduced to $O(\log k + \log\log n)$ from $O(k + \log n)$. We use two techniques for the reduction: (i) As in [8], 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 $O(\log k)$ since 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) [10] that represents a large number as a set of small numbers. In particular, we use the technique called Chinese Remainder Theorem (CRT) [11]. The CRT says that for two positive integers n_1 and n_2 ($n_1, n_2 < n$), if the remainders of the integers when divided by each of the first $\log n$ prime numbers $2, 3, 5, \ldots, U$ are the same, then $n_1 = n_2$ holds [11]. The prime number theorem guarantees that the ($\log n$)-th prime U satisfies $U = O(\log^2 n)$. Thus, agents compare distances between adjacent active nodes 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 IDs 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. 3). 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 ID as the remainder divided by the l-th prime $prime_l$. In Fig. 3, when $prime_l = 3$, a_i gets its ID $ID_i = (2, 1)$.

During the traversal, a_i lexicographically compares its ID ID_i with IDs 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 IDs a_i ever found 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

Algorithm 2 The behavior of active agent a_i in the selection phase

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

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 $(l+1)$-th iteration using the next prime $prime_{l+1}$. (c) If $same = false$, they terminate the current subphase. If a_i has the maximum ID, 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*.

Pseudocode is described in Algorithm 2. Each agent a_i uses boolean variable *max* ($max = true$ means ID_i is the maximum among IDs a_i has ever found).

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 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[6]. 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)[7]. On

[6] When an 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.

[7] By the atomicity of an action, when an agent moves to some leader node, the leader agent already starts its collection phase and leaves the leader node.

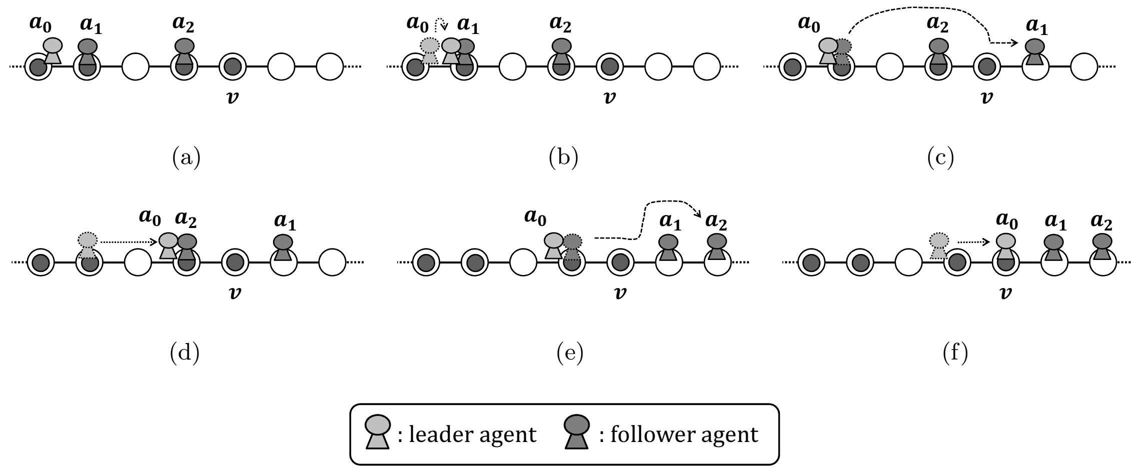


Fig. 4. An example of the collection phase ($fNum = 2$).

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, and the agents in a group are deployed at consecutive nodes starting from a base node.

For example, in Fig. 4 there exists one leader agent a_0 and two follower agents a_1 and a_2 between a_0 and its adjacent 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.

4.3 Deployment phase

In this phase, leader agents control follower agents so that they should move to and stay at their target nodes to achieve uniform deployment. 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 $a_1, a_2, \ldots, a_{fNum}$ are following a_0 in this order. At the beginning of the first subphase, they stay at consecutive nodes. Each subphase consists of $fNum$ iterations. In the l-th iteration, each of the l agents $a_{fNum-l+1}, a_{fNum-l+2}, \ldots, a_{fNum}$

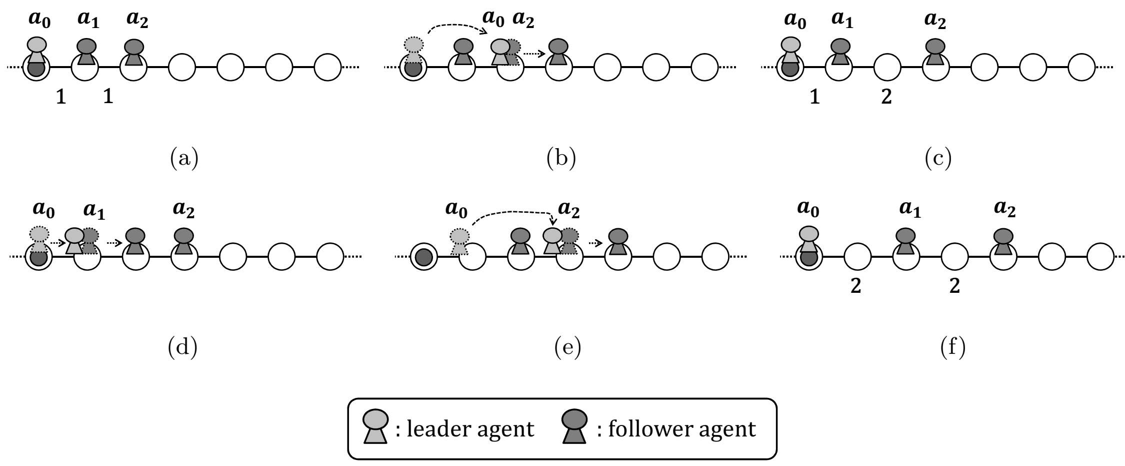


Fig. 5. An example of the deployment phase (*fNum* = 2).

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-the iteration is realized as follows. Leader agent a_0 firstly moves to the node where $a_{fNum-l+1}$ is staying and stays there until $a_{fNum-l+1}$ moves to the next node. Then, a_0 moves to the node where $a_{fNum-l+2}$ is staying to make $a_{fNum-l+2}$ 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. 5 there exist one leader agent a_0 and two follower agents a_1 and a_2 (i.e., *fNum*=2). Let d_1 (resp., d_2) be the distance from a_0 to a_1 (resp., a_1 to a_2). In (a), $d_1 = d_2 = 1$ 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, $d_1 = 1$ and $d_2 = 2$ hold. If 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 $(fNum - 1)$-th 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, $d_1 = d_2 = 2$ holds and a_i determines its next behavior depending on the location of agents. Each leader repeats such a behavior until it recognizes that the locations of agents are uniformly deployed.

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 with* $O(\log k + \log\log n)$ *memory space per agent,* $O(n^2 \log n)$ *time, and* $O(kn^2 \log n)$ *total number of moves.* □

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 $O(k + \log n)$ 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 with $O(\log k + \log\log n)$ memory space per agent, $O(n^2 \log n)$ time, and $O(kn^2 \log n)$ total number of moves.

As a future work, for agents without multiplicity detection we want to propose a space-optimal (i.e., $O(\log n)$ memory) algorithm to solve the problem. Also, for agents with weak multiplicity detection we want 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.

References

1. R. S. 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. D.B. Lange and M. Oshima. Seven good reasons for mobile agents. CACM, 42(3):88–89, 1999.
3. E. Kranakis and D. Krizanc. An algorithmic theory of mobile agents. International Symposium on Trustworthy Global Computing, Vol. 4661. pages 86-97, 2006.
4. J. Cao, Y. Sun, X. Wang, and S.K. Das. Scalable load balancing on distributed web servers using mobile agents. JPDC, 63(10):996–1005, 2003.
5. P. Flocchini, G. Prencipe, and N. Santoro. Self-deployment of mobile sensors on a ring. Theoretical Computer Science, 402(1):67–80, 2008.
6. E. Yotam and B. M. Alfred. Uniform multi-agent deployment on a ring. Theoretical Computer Science, 412(8):783–795, 2011.
7. 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.
8. M. Shibata, T. Mega, F. Ooshita, H. Kakugawa, and T. Masuzawa. Uniform deployment of mobile agents in asynchronous rings. PODC, pages 415–424, 2016.
9. G. Tel. Introduction to distributed algorithms. Cambridge university press, 2000.
10. OR. Amos and P. Benjamin. Residue number systems: theory and implementation, volume 2. World Scientific, 2007.
11. D. Pei, A. Salomaa, and C. Ding. Chinese remainder theorem: applications in computing, coding, cryptography. World Scientific, 1996.