How to Simulate Message-Passing Algorithms in Mobile Agent Systems with Faults

Tsuyoshi Gotoh[1(✉)], Fukuhito Ooshita[2], Hirotsugu Kakugawa[1], and Toshimitsu Masuzawa[1]

[1] Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
{t-gotoh,kakugawa,masuzawa}@ist.osaka-u.ac.jp
[2] Nara Institute of Science and Technology, 8916-5 Takayamacho, Ikoma, Nara 630-0101, Japan
f-oosita@is.naist.jp

Abstract. We propose a fault-tolerant algorithm to simulate message-passing algorithms in mobile agent systems. We consider a mobile agent system with k agents where f of them may crash for a given f $(\leq k-1)$. The algorithm simulates a message-passing algorithm, say Z, with $O((m+M)f)$ total agent moves where m is the number of links in the network and M is the total number of messages created in the simulated execution of Z. The previous algorithm [5] can tolerate $k-1$ agent crashes but requires $O((m+nM)k)$ total agent moves. Therefore, our algorithm improves the total number of agent moves for $f = k-1$ and requires a smaller number of total moves if f is smaller.

1 Introduction

A *distributed system* is composed of many computers (nodes) that can communicate with each other. Recently distributed systems have become larger, which makes it difficult to design them. As a paradigm to circumvent the difficulty, *mobile agents (agents)* have attracted a lot of attention [3]. An agent is a software program which can move autonomously in a distributed system, collect information at visited nodes, exchange the information with other agents and execute actions at visited nodes using the information. An agent can be considered as encapsulation of data and actions, and the number of agents in a network restricts concurrency of actions executed in the network. This makes algorithm design easier in mobile agent systems than in message-passing systems. So far many agent-based algorithms have been proposed for several tasks, such as leader election, naming, locating agents, rendezvous, stabilization, termination detection, exploring and topology recognition [3]. From the viewpoint of security, algorithms for intruder capture [1,2] and network decontamination [10,12] have been proposed.

While most works stated above assume agents and nodes work correctly, recent large-scale distributed systems can no longer make such an assumption. For this reason, some researches consider faulty nodes where their states are disrupted [6,7] or visiting agents are destroyed [8,11]. In addition, we should consider the scenario such that agents themselves become faulty. For example, if the system spreads to all over the world, agents may move a long distance by passing through lots of physical links. During the movement, agents may crash (or disappear) when one of the links suffers from an error. Hence algorithms tolerant to faults of agents are required for many tasks.

As an approach to realize agent-based algorithms for many tasks, we focus on simulation of *message-passing algorithms* in mobile agent systems [4,5,13]. If agents can simulate message-passing algorithms efficiently, they can efficiently execute many tasks which are suitable for message-passing algorithms rather than mobile agent algorithms. Moreover, from the viewpoint of algorithm designing, it is more efficient to design a simulation algorithm of message-passing algorithms by mobile agents because efficient message-passing algorithms have been proposed for many tasks in literature [9,14]. The only existing work to simulate message-passing algorithms in a fault-tolerant manner is the one by Das et al. [5]. In this work, the authors propose two algorithms to simulate message-passing algorithms by asynchronous agents when at most $k-1$ agents crash, where k is the number of agents. One algorithm simulates a message-passing algorithm with $O((m+nM)k)$ total agent moves by agents with distinct IDs, where m is the number of links, n is the number of nodes and M is the number of messages created in the simulated execution of the message-passing algorithm. Another algorithm simulates a message-passing algorithm with $O((m+nk)M)$ total moves by anonymous agents. Note that, in the algorithm for agents with distinct IDs, the number of moves per message is (or the multiplication factor of M) $O(nk)$.

In this paper, we propose a new *fault-tolerant algorithm* to simulate message-passing algorithms by asynchronous agents with distinct IDs. Our algorithm assumes at most f agents crash for a given $f \leq k-1$, and simulates a message-passing algorithm with $O((m+M)f)$ total agent moves. That is, the number of moves per message is $O(f)$ when $M = \Omega(m)$ holds. Note that because f agents can become faulty and agents move asynchronously (i.e., the time required to move along a link is unbounded and unpredictable), every message should be delivered by $f+1$ agents in the worst case. This means our algorithm is asymptotically optimal concerning of the number of agent moves per message.

Our algorithm improves the previous algorithm [5] in the number of agent moves. The improvement is achieved by adopting the depth-first simulation while the previous one adopts the breadth-first one. More precisely, the previous algorithm simulates the synchronous execution of a message-passing algorithm. To realize a synchronous round, each agent traverses the network to find messages to transfer, which requires $O(n)$ redundant moves per message in the worst case. To avoid such redundant moves, our algorithm traces a message to find another message to transfer. That is, our algorithm allows each agent to deliver messages in the depth-first fashion; when an agent visits a node with carrying a message (to be delivered to the node) and finds another message to transfer in the node, it takes the message and transfers it to the destination node. Note that these two simulation algorithms simulate different executions of the message-passing algorithm, each of which is a possible execution.

Due to the space constraint, most of the proofs have been omitted from this paper and can be found in the appendix.

2 Preliminaries

2.1 Network

A network is modeled by a connected undirected graph $G = (V, E)$, where V is a set of nodes and E is a set of communication links. Each link $e \in E$ connects distinct nodes in V. A link that connects nodes u and v is denoted by e_{uv} or e_{vu}. In this paper, we denote $n = |V|$ and $m = |E|$. The degree of u is defined as the number of incident links of u, and is denoted by deg_u. The maximum degree $\max\{deg_u \mid u \in V\}$ of the network is denoted by Δ. The neighbors of u are nodes directly connected to u, and the set of them is denoted by N_u. Each link incident to node u is locally labeled at u by bijection $\lambda_u : \{(u, v) : v \in N_u\} \to \{1, 2, \ldots, deg_u\}$ and u distinguishes its neighbors by the labels. Note that, $\lambda_u(e_{uv}) \neq \lambda_u(e_{uw})$ holds for distinct neighbors v and w of u. The labeling is independent from those of other nodes; for an edge e_{uv}, $\lambda_u(e_{uv}) \neq \lambda_v(e_{uv})$ may hold. We say $\lambda_u(e_{uv})$ is a port number (or port) of e_{uv} on u.

We consider two different computation models of a network, a message-passing model and a mobile agent model, which are defined in the following subsections and follow [5].

2.2 Message-Passing Model

In the message-passing model, each node u is modeled as a state machine (S_u, δ_u), where S_u is a set of (possibly infinite) node states and δ_u is a state transition function. The state machine may be dependent on its node ID if exists: nodes with different IDs may be modeled as different state machines. Two states in S_u are designated as initial states: one is for an (spontaneous) initiator and the other is for a non-initiator. The transition function δ_u is defined as $\delta_u : S_u \times M \times P \to S_u \times 2^{M \times P}$, where M is a set of all possible messages (including a special null message) and P is a set of port numbers. The function δ_u determines, from a current state and a received message with its incoming port, a subsequent state of the node and a set of messages to be sent with their outgoing ports. The initial state for the initiator and the special message null are used only for the first action of the initiator, which is independent of the incoming port of the null message. The state machine can depend on the degree and the ID (if exists) of the node.

Each node executes the following operations atomically in each step: (1) it receives a message or initiates an algorithm spontaneously, (2) executes local computation and updates its own state, and (3) if necessary, sends messages to its neighbors by using the primitive $SEND(c, \lambda_u(e_{uv}))$ repeatedly (node u can send a message c to node v by using the primitive $SEND(c, \lambda_u(e_{uv}))$). There exists at least one spontaneous initiator, which is assigned the special initial state and initiates an algorithm spontaneously (by receiving the null message). Except for the initial steps of initiators, every process takes a step only when it receives a message. Note that, since the set of initiators is unknown in advance, algorithms should work correctly for any set of initiators.

Communication in the message-passing model is reliable, that is, it satisfies the following:

[A1] Every message sent by node u to its neighbor v is eventually received by v exactly once.
[A2] A message is received by node u only when it was previously sent to u by neighbor v.

Each link in the network is FIFO, that is, when u sends messages c_1 and c_2 to v in this order, v receives c_1 before c_2. The system is asynchronous, that is, the time required to transfer a message between neighbors is finite but unbounded.

2.3 Mobile Agent Model

In the mobile agent model, all the actions (i.e., computation and communication) on a network are carried out by mobile agents. Let A be a nonempty set of mobile agents existing on the network and $k = |A|$. Each agent has its own memory, called a *notebook*. In this model, a node works only as a repository and a memory on a node is called a *whiteboard*.

Each agent a has a unique ID $a.id$ and we assume each ID is represented in $O(\log k)$ bits. Every agent do not know n. Each agent a is initially allocated to some node called a *homebase* of a. We assume $k \leq n$ and homebases of agents are mutually distinct.

Each agent a is modeled as a state machine (S_a, δ_a), where S_a is a set of agent states and δ_a is a state transition function. A state in S_a is designated as an initial state. The transition function δ_a is defined as $\delta_a : S_a \times W \times (P \cup \{0\}) \rightarrow S_a \times W \times (P \cup \{0\})$, where W is the set of all possible whiteboard states and P is the set of port numbers. The inputs of the transition function δ_a are a current state of an agent, a state of the whiteboard on its current node, and a port number (or 0 explained in the below) through which the agent arrived, and the outputs are a subsequent state of the agent, a new state of the whiteboard, and a port number (or 0 explained in the below) through which the agent leaves. Port number 0 in the inputs implies the agent initiates the algorithm at its current node or the agent stays at the current node from its previous action, while $p > 0$ implies the agent arrives at the current node from port p. Port number 0 in the outputs implies the agent stays at the current node, and $p > 0$ implies the agent leaves the current node through port p. The state machine can depend on the agent ID. The state transition can depend on the degree and the ID (if exists) of the node which the agent is staying at or visiting. This is implemented by storing the node degree and ID in the whiteboard.

Each agent executes the following operations atomically in each step: (1) It arrives at a node or initiates an algorithm at its homebase, (2) executes local computation and updates its own state (including its notebook contents) and the whiteboard contents of its current node, and (3) moves to a neighbor of its current node or stays at its current node.

This paper considers simulation of the message-passing model on the mobile agent model. We assume that the target model (or the message-passing model) is reliable but the host model (or the mobile agent model) is prone to faults. An agent may crash (or disappear) when it moves through a link, but it never crashes when it is on a node. We assume at most $f \leq k-1$ agents crash and every agent knows the upper bound f. After an agent leaves a node, it arrives at the next node eventually unless it crashes during the movement. Once an agent crashes, it disappears from the network forever. We say an agent is faulty (resp., non-faulty) if it crashes (resp., never crashes) during the execution. Note that agents cannot recognize faulty agents as long as they work correctly. Each link in the network is FIFO, that is, when agents a_1 and a_2 move from node u to node v in this order, a_1 arrives at v before a_2 unless a_1 crashes during the movement. The system is asynchronous, that is, the time required for an agent to move from a node to its neighbor is finite but unbounded.

3 Agent-Based Simulation of Message-Passing Algorithms

In this section, we first consider, as target algorithms of agent-based simulation, message-passing algorithms with a *finite* number of messages. We present a simulation algorithm, correctness proof and analysis of move and memory complexity in Sect. 3.1. We denote Z as the simulated message-passing algorithm for hereafter. A message-passing algorithm that eventually terminates uses a finite number of messages and is a target algorithm of the simulation algorithm in Sect. 3.1. Thus, most of algorithms designed so far can be the target of the simulation [9,14]. In Sect. 3.2, we briefly present the simulation algorithm targeting message-passing algorithms with an *infinite* number of messages.

3.1 A Case of a Finite Number of Messages

The Execution of a Simulation Algorithm. In this subsection, we propose an agent-based simulation algorithm of a message-passing algorithm with a finite number of messages (i.e., an eventually terminating algorithm). Our algorithm consists of two parts, (1) searching initiators (search part) and (2) simulating an execution of nodes and delivering messages (delivery part).

First, we present the search part, (1) searching initiators. Each agent starts to search initiators from its homebase by the depth-first search. When it finds an initiator, it starts the delivery part, (2) simulating an execution of nodes and delivering messages. After completing the delivery part, it resumes the search part to find another initiator. The agent records its searching path of the search part by writing the incoming port in the whiteboard of each visited node so that it can backtrack.

In the search part, an agent backtracks to the previous node when at least one of following conditions is satisfied.

1. There is no unsearched port at the current node.
2. A cycle is detected in its searching path of the search part.
3. The agent detects that other $f+1$ agents which execute the search part have already visited the current node.

Conditions 1 and 2 come from the depth-first search. Condition 3 is introduced to save the total number of agent moves. Our algorithm can tolerate agent crashes by using multiple agents to transfer a message, however it is enough that each message is transfered by $f+1$ agents since at most f agents crash (there is at least one non-faulty agent in $f+1$ agents). Thus, an agent backtracks when it detects that other $f+1$ agents which execute the search part. The agent terminates its execution when it completes the search part and returns to its homebase.

Next, we present the delivery part, (2) simulating an execution of nodes and delivering messages.

An agent starts the simulation when it finds an initiator during the depth-first search of the first part. Note that, by Condition 3 of the first part, at most $f+1$ agents visit each initiator and start simulation.

The agent delivers messages successively in the depth-first fashion, that is, if there exists a message to transfer to another node in the node that the agent visits to deliver a message, it takes a message from the node and delivers it. The agent records its delivering path in the same way as the search part so that it can backtrack.

Since a message is transfered by at most $f+1$ agents for fault-tolerance, the message may be delivered multiple times. However it is processed only once, that is, an agent simulates the action of a node on receipt of a message only when the message has not been simulated.

When an agent takes a message from the node, the agent stores its ID to *send-member* of the message in the whiteboard of the node to indicate that the agent transfered the message. The message is deleted from the node when one of its *send-member* agents returns and, at this time, *send-member* of the current node is reset to empty.

In the delivery part, an agent backtracks to the previous node when at least one of following conditions is satisfied.

1. There is no message to transfer at the current node.
2. A cycle is detected in its delivering path of the delivery part.
3. The current node is locked using the port other than the one the agent arrives through. We describe the locking mechanism later.
4. The current node is locked but the agent is not a *lock-member* agent of the node when the agent backtracks to the node.

Condition 1 realizes message deliveries in the depth-first fashion. Condition 2 is introduced to prevent the delivering path from growing so long, which saves the whiteboard space of nodes. Conditions 3 and 4 are introduced to guarantee that all the messages are delivered since using only Conditions 1 and 2 makes some messages remain undelivered as explained below.

Consider the case of Fig. 1. First, agent a arrives at t and delivers messages from t to u and agent b follows a and arrives at u. Then, a backtracks to t and deletes messages at t and v while b is still in transit in link e_{uv}. Second, an agent c arrives at y from x and delivers messages from y to v through z and w. Then, c crashes after generating two messages at v, one is to y and the other is to u in this order. After that, agent b arrives at v from u and delivers messages from v to v through y, z and w. Then, b detects a cycle at v, backtracks to w, and deletes the message from w to v. Then, while b backtracks from w to z, b crashes. Here, node v has a message to transfer to u but it is possible that no agent arrives at v after the situation since there is no message toward v. Thus, in this case, the message from v to u may be left undelivered.

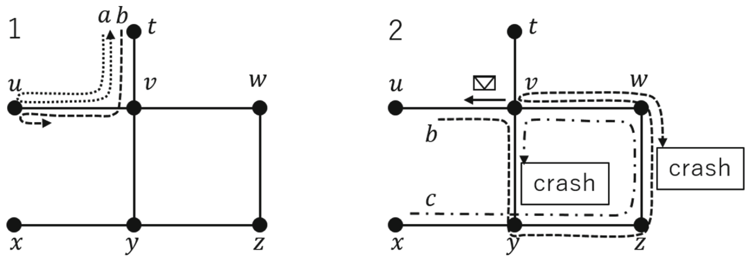

Fig. 1. An example where Conditions 1 and 2 allow a message to remain undelivered.

A possible way to avoid such undelivered messages is not to introduce Condition 2. In this case, an agent continues to deliver messages as long as the current node has messages to transfer. But this allows the delivering path to become so long when a long message chain exists. It requires large whiteboard spaces since the delivering path is recorded in the whiteboards of nodes. Thus, we insist on Condition 2 to save the whiteboard space. So we introduce the locking of nodes as another way to guarantee deliveries of all messages.

A reason why the above case happens is that agents which have distinct delivering paths deliver the same message. So we design the locking so that it prevents distinct delivering paths from merging.

An agent locks the current node by writing, to the whiteboard, the port through which it arrives when the current node is unlocked. An agent that arrives at the locked node delivers a message from the node only when it arrives through the port that is used for the locking. Otherwise, it has to backtrack to the previous node in the delivering path. Note that, since all the delivering paths of the delivery part of agents start from an initiator, by repeating above, agents which deliver the same message must have the same delivering path.

An agent stores its ID to *lock-member* in the whiteboard of a locked node when the agent locks the node or arrives through the port that is used for the locking. When a *lock-member* agent backtracks from the locked node, it unlocks the node and resets *lock-member* of the node to empty.

Condition 4 makes an agent backtrack to the previous node in the delivering path when it backtracks to a node but is not a *lock-member* agent of the node. This implies that the node was already unlocked for the locking such that the agent was a *lock-member* agent, that is, an agent which delivered the message from the node may have a distinct delivering path. This makes the agent keep backtracking along the delivering path until the agent reaches a node where the agent is a *lock-member* or it started the simulation of nodes and delivering messages.

An agent resumes the message delivery when it finds its ID in *lock-member*. It terminates the delivery part and resumes the search part (i.e., searching an initiator) when it reaches the starting node but is not a *lock-member* agent.

The Pseudo Codes. Algorithms 1, 2, 3 and 4 are the pseudo codes of the fault-tolerant simulation algorithm.

We use operations $enqueue(q, M)$, $dequeue(q)$ and $head(q)$ to handle message queue q at a node. Operation $enqueue(q, M)$ for message sequence M is used to append M to the tail of q, $dequeue(q)$ is used to delete the head element of q and $head(q)$ is used to refer to the head element of q. Notation $v.var$ denotes variable var stored in the whiteboard of the current node v, and $a.var$ denotes variable var stored in the notebook of agent a.

We show the variables with initial values and their types in Table 1.

In Table 1, we denote $v.port_{unsearched}$, $v.parent_{search}$, $v.parent_{transmit}$ and $v.receive$ as a set of pairs but, for convenience, we use them as the arrays (e.g., $v.parent_{search}[a.id]$) in pseudo code and explanation below.

At the moment agent a starts Algorithm 1 at node v, a adds 0 to $v.parent_{search}[a.id]$ to declare that v is the homebase of a and adds $\{1, \ldots, deg_v\}$ to $v.port_{unsearched}[a.id]$. Then, a starts the depth-first search with recording the port through which a arrives in $v.parent_{search}[a.id]$ at each visited node v. When a finds an initiator, a executes $Transmit()$ (Algorithm 2) to simulate the message-passing algorithm Z. For saving whiteboard space, if the current node's $v.parent_{search}[a.id]$ is not $\perp$ (it means v is included in the path of a), a backtracks to the previous node. For decreasing the number of movements, a also backtracks to the previous node if the current node's $v.parent_{search}$ has $f + 1$ (it means that $f + 1$ agents have already visited the node during the search part of the agents) IDs. Agent a terminates if the current node's $v.parent_{search}[a.id]$ is 0 (it means v is the homebase of a) and there is no unsearched port.

At the moment agent a starts $Transmit()$, a adds 0 to $v.parent_{transmit}[a.id]$ to declare that v is the starting node of $Transmit()$. Then, a transfers messages successively in the depth-first fashion with recording the port through which a arrives in $v.parent_{transmit}[a.id]$ and its ID in $v.send\_member$ at each visited node. For saving whiteboard space, if the current node's $v.parent_{transmit}[a.id]$

Table 1. Variables used in the pseudo codes

	Name	Initial value type	What its value means
Node v	$v.port_{unsearched}$	$\emptyset$	A pair of (a, P) in the set implies port sets P of v is unsearched for agent a in the first part (i.e., searching an initiator)
		Set of $(agentID, ports)$	
	$v.parent_{search}$	$\emptyset$	A pair (a, p) in the set implies that agent a arrives at v for the first time from port p in the first part
		Set of $(agentID, port)$	
	$v.parent_{transmit}$	$\emptyset$	The same as $v.parent_{search}$ but for the second part (i.e., simulating an execution of nodes and delivering messages)
		Set of $(agentID, port)$	
	$v.init$	True or false	Indicates whether v is an initiator of the target (message-passing) algorithm it is true only if v is an initiator
		Boolean	
	$v.port_{lock}$	$\perp$	Port p implies that v is locked using p, and $\perp$ implies that v is unlocked
		Port or $\perp$	
	$v.send$	Empty sequence	Messages to transfer to neighbors
		Message queue	
	$v.send\_member$	$\emptyset$	An ID set of agents that are *send-member* of the head message of $v.send$
		Set of agentIDs	
	$v.lock\_member$	$\emptyset$	An ID set of agents that are *lock-member* of v
		Set of agentIDs	
	$v.state_n$	The initial state	v's state of the target algorithm
		State	
	$v.receive$	$\emptyset$	The latest messages v received from each port
		Set of $(port, message)$	
Agent a	$a.msg$	$\perp$	A message which a is delivering
		Message	

Algorithm 1. main

```
1: $v.port_{unsearched}[a.id] \leftarrow \{1, \cdots, deg_v\}$
2: $v.parent_{search}[a.id] \leftarrow 0$;
3: **while** (1)
4:   **if** $(v.init = true) \vee (v.port_{lock} = 0)$ **then** //the current node is initiator
5:     $Transmit()$;
6:   **if** $(v.port_{unsearched}[a.id] \neq \emptyset)$ **then** //search an unsearched port
7:     $v.port_{unsearched}[a.id] \leftarrow v.port_{unsearched}[a.id]/\{p\}$;
8:     move through $p$;
9:     arrive from $q$;
10:    **if** $(|v.parent_{search}| = f+1)$ **then** //the current node is visited by $f+1$ agents
11:      move through $q$ (return to the previous node);
12:    **else**
13:      **if** $(v.parent_{search}[a.id] \neq \perp)$ **then** //find $a$'s own ID
14:        $v.port_{unsearched}[a.id] \leftarrow v.port_{unsearched}[a.id]/\{q\}$
15:        move through $q$ (return to the previous node);
16:      **else** //arrive at $v$ for the first time
17:        $v.parent_{search}[a.id] \leftarrow q$;
18:        $v.port_{unsearched}[a.id] \leftarrow \{1, \cdots, deg_v\}/\{q\}$
19:   **else** //there is no unsearched port
20:     $p \leftarrow v.parent_{search}[a.id]$;
21:     **if** $(p = 0)$ **then**
22:       $break$;
23:     **else**
24:       move through $p$(return to the previous node);
25: **end while**
```

is not $\perp$ (it means v is included in the delivering path of a), a backtracks to the previous node. Agent a also backtracks to the previous node when v is locked using a port other than the one a arrives through, that is, $v.port_{lock}$ of the current node is not $\perp$ and the other than the one a arrives through. Agent a terminates $Transmit()$ if the current node's $v.parent_{transmit}$ is 0 (it means a starts $Transmit()$ from v) and there is no message in the message queue of v. The message is deleted from the node when one of its $v.send\_member$ agents backtracks. Agent a deletes the messages which a delivered and resets $v.send\_member$ to $\emptyset$ if $a.id$ is included in $v.send\_member$ when returning to v.

Agent a transfers messages left in v if there is $a.id$ in $v.lock\_member$ when a backtracks to v (it means a is *lock-member* of v) on $Transmit()$. It stores $a.id$ in $v.lock\_member$ when a arrives at v with a message and v is not locked or a arrives through the port used for locking. If there is not $a.id$ in $v.lock\_member$ at the current node when a backtracks to v, a executes $Go\_back()$ until a finds $a.id$ in $v.lock\_member$. If $Go\_back()$ outputs 0, a terminates $Transmit()$ and resumes Algorithm 1. If $Go\_back()$ outputs 1, a continues $Transmit()$ to transfer messages.

Function $Go\_back()$ (Algorithm 3) is called in $Transmit()$ when a backtracks to the previous node. Agent a continues to backtrack through the port

Algorithm 2. $Transmit()$

```
1: $Process(\emptyset, \emptyset)$;//process a unprocessed initiator
2: if ($v.port_{lock} = \perp$) then
3:    $v.port_{lock} \leftarrow 0$;
4: $v.lock\_member \leftarrow v.lock\_member \cup \{a.id\}$;
5: $v.parent_{transmit}[a.id] \leftarrow 0$;//mark 0 on the starting node of $Transmit()$
6: while (1)
7:    if ($v.send \neq \emptyset$) then
8:       $a.msg \leftarrow head(v.send)$;//copy the head message of $v.send$
9:       $v.send\_member \leftarrow v.send\_member \cup \{a.id\}$;
10:      move through the destination port $p$ of $a.msg$;
11:      arrive from $q$;
12:      $Process(a.msg, q)$;
13:      $a.msg \leftarrow \perp$;
14:      if $((v.port_{lock} = \perp) \vee (v.port_{lock} = q)) \wedge (v.parent_{transmit}[a.id] = \perp)$ then //$v$ is not locked or locked by the incoming port, and $v$ is not included in $a$'s path
15:         if ($v.port_{lock} = \perp$) then
16:            $v.port_{lock} \leftarrow q$;
17:         $v.lock\_member \leftarrow v.lock\_member \cup \{a.id\}$;
18:         $v.parent_{transmit}[a.id] \leftarrow q$;
19:      else
20:         if ($Go\_back(q) = 0$) then $return$;//backtrack to a node s.t. $a$ is a $lock$-$member$
21:   else
22:      if ($a.id \in v.lock\_member$) then
23:         $v.port_{lock} \leftarrow \perp$;
24:         $v.lock\_member \leftarrow \emptyset$;
25:      $q \leftarrow v.parent_{transmit}[a.id]$;
26:      $v.parent \leftarrow \perp$;
27:      if ($q = 0$) then
28:         $return$;
29:      else
30:         if ($Go\_back(q) = 0$) then $return$;//backtrack to a node s.t. $a$ is a $lock$-$member$
31: end while
```

in $v.parent_{transmit}[a.id]$ until a finds $a.id$ in $v.lock\_member$. If a finds $a.id$ in $v.lock\_member$, $Go\_back()$ outputs 1 and a restarts $Transmit()$ to transfer messages from the node. If a does not find $a.id$ in $v.lock\_member$, $Go\_back()$ outputs 0 and a terminates $Transmit()$ and resumes the depth-first search. While searching $a.id$, a removes the messages which it delivered.

Function $Process()$ (Algorithm 4) is used to simulate an execution of nodes in Z. If the current node is an unprocessed initiator, a simulates an execution of the node. To simulate the execution of an initiator, a uses $simulate(v.state_n, \perp)$ and it gets a new node state s and a new message sequence M. To simulate the execution of a node on receipt of a message c, a uses $simulate(v.state_n, c)$ and it gets a new node state s and a new message sequence M.

Algorithm 3. *Go_back*(q)

```
1: move through $q$ (return to the previous node);
2: while (1)
3:   if ($a.id \in v.send\_member$) then //remove the message which $a$ transfered
4:     $v.send\_member \leftarrow \emptyset$;
5:     $dequeue(v.send)$;
6:   if ($a.id \in v.lock\_member$) then //if $a$ is a $v.lock\_member$ agent, restart to send messages
7:     $return$ 1;
8:   else
9:     $q = v.parent_{transmit}[a.id]$;
10:    $v.parent_{transmit}[a.id] \leftarrow \perp$;
11:    if ($q = 0$) then //the starting node of $Transmit()$
12:      $return$ 0; //return from $Transmit()$
13:    else //if $a$ is not a $v.lock\_member$ agent, return to the previous node
14:      move through $q$ (return to the previous node);
15: end while
```

Each message may be delivered multiple times by some agents on Algorithm 2. To make sure that each message is processed once, the latest message which are delivered from each port p is stored in $v.receive[p]$ and a message is not processed if it is already recorded in $v.receive[p]$.

Algorithm 4. *Process*(c, q)

```
1: if ($v.init = true$) then
2:   $v.init \leftarrow false$;
3:   $(s, M) \leftarrow simulate(v.state_n, \perp)$;
4:   $v.state_n \leftarrow s$;
5:   $enqueue(v.send, M)$;
6: if ($c \neq \perp$) $\wedge$ ($c \neq v.receive[q]$) then //simulate the process of an initiator and of a node which receive $c$ from $q$
7:   $v.receive[q] \leftarrow c$;
8:   $(s, M) \leftarrow simulate(v.state_n, c)$;
9:   $v.state_n \leftarrow s$;
10:  $enqueue(v.send, M)$;
```

Proof of Correctness. In this part, we show that the proposed algorithm simulates Z correctly.

First, we define the time instants of send and receive operations in the simulation of message-passing algorithm Z.

- The time instant that v sends message c in the simulation of Z is defined as the time instant that an agent stores c to $v.send$.
- The time instant that v receives message c in the simulation of Z is defined as the time instant that an agent with message c arrives at v and simulates local computation of v initiated by receipt of c for the first time.

Hereafter, we say an agent is in the delivery mode when it executes procedure $Transmit()$, procedure $Go\_back()$ or procedure $Process()$, and an agent is in the search mode otherwise. The following lemmas hold.

Lemma 1. *By the proposed algorithm, each node is visited by at least one non-faulty agent of the search mode and hence every initiator starts execution of Z.*

Lemma 2. *During the execution of the proposed algorithm, there is no message in $v.send$ if $v.port_{lock}$ is $\perp$.*

Lemma 3. *By the proposed algorithm, agents simulate reliable communication.*

Lemma 4. *By the proposed algorithm, agents simulate the FIFO order of message communication.*

From Lemmas 1, 3 and 4, the proposed algorithm initiates execution of all initiators and delivers all messages to their destinations correctly. This implies the following theorem holds.

Theorem 1. *The proposed algorithm simulates Z correctly when at most $f \leq k-1$ agents are faulty.*

Evaluation. In this part, we evaluate the move complexity of agents. Clearly it depends on the target message-passing algorithm Z. Let M and L be the number and the maximum size of messages created in the simulated execution of algorithm Z respectively.

Theorem 2. *The proposed algorithm simulates Z with $O((m+M)f)$ total agent moves, $O(L+\log k)$ agent memory and $O((M+\Delta)L + f\Delta\log(k\Delta))$ additional node memory.*

Proof. We show only the evaluation of the total agent moves. For the search mode, at most $f+1$ agents search each link in two directions and one search consists of a forward move and a backward move. Thus, the move complexity of the search mode is $2 \cdot 2 \cdot m \cdot (f+1) = 4m(f+1)$. For the delivery mode, at most $f+1$ agents carry each message of Z by forward moves, and every agent makes one backward move for each forward move. Thus, the total move complexity of the simulation mode is $2M(f+1)$. Thus, the move complexity is $4m(f+1) + 2M(f+1) = O((m+M)f)$. □

3.2 A Case of an Infinite Number of Messages

The simulation algorithm we propose in Sect. 3.1 can not simulate message-passing algorithms Z with an infinite number of messages as explained below.

Consider the case of Fig. 2. There are two independent infinite circulations of messages C_a and C_b. If all the agents in the network transfer the messages

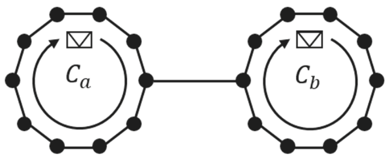

Fig. 2. An example where algorithm proposed in Sect. 3.1 cannot simulate Z with an infinite number of messages.

included in C_a in the depth-first fashion, the messages included in C_b are not delivered forever.

To simulate Z with an infinite number of messages, we introduce, to the depth-first message delivery, restriction on the number of message deliveries and change the algorithm as follows.

1. Instead of the depth-first message delivery in Sect. 3.1, the depth-first delivery with restricted ℓ messages is adopted, which is a modification of the depth-first delivery such that an agent backtracks when the number of messages which the agent delivered reaches ℓ.
2. Each agent repeats the depth-first search of the search part infinitely and traverses the whole network during the depth-first search of the search part.
3. An agent of the search mode stops execution of the simulation algorithm when it finds $f + 1$ delivery mode agent names in the current node.
4. An agent starts the delivery part not only when it finds an initiator, but also when it finds a message to transfer on an unlocked node.

We show that the number of agent moves per message is $O(f)$ in the modified algorithm. First, the following lemmas hold.

Lemma 5. *During the execution of the modified algorithm, there remains at least one non-faulty agent.*

Lemma 6. *By the modified algorithm, at least ℓ messages are delivered during the depth-first search of the search part of an agent.*

In the modified algorithm, every message is transfered by at most $f+1$ agents as in the algorithm in Sect. 3.1. From Lemma 6, at least ℓ messages are delivered during the depth-first search of the search part of an agent, which takes m agent moves in the first search and n agent moves in the second or later search. That is, at least ℓ messages are delivered during $kn+f\ell$ ($km+f\ell$, for the first depth-first search) agent moves. Since an agent traverses the whole network, the agent can get k and n in the first depth-first search of the search part. With setting ℓ to be kn, the number of agent moves per message becomes $O(f)$.

Theorem 3. *The modified algorithm simulates Z with $O(f)$ agent moves per message.*

4 Conclusion

In this paper, we proposed a new algorithm to simulate a message-passing algorithm in the mobile agent model. It requires $O((m + M)f)$ agent moves to tolerate when at most $f \leq k - 1$ agents crash where m is the number of links in the network and M is the number of messages in the simulated execution of the message-passing algorithm. The previous algorithm requires $O((m + nM)k)$ agent moves when at most $k - 1$ agents crash. Thus, our algorithm improves the previous algorithm in the number of agent moves. Furthermore, we proposed a simulation algorithm for message-passing algorithms with an infinite number of messages. It also improves the number of agent moves per message to $O(f)$ from $O(nk)$.

Our algorithm and the previous algorithm [5] simulate different executions of the message-passing algorithm. The actual number of agent moves depends on the number and the creation pattern of messages in the simulated execution. Thus, it is interesting as a future work to investigate the actual number of agent moves for concrete examples of message-passing algorithms.

References

1. Barrière, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Capture of an intruder by mobile agents. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 200–209. ACM (2002)
2. Blin, L., Fraigniaud, P., Nisse, N., Vial, S.: Distributed chasing of network intruders. In: Flocchini, P., Gąsieniec, L. (eds.) SIROCCO 2006. LNCS, vol. 4056, pp. 70–84. Springer, Heidelberg (2006). doi:10.1007/11780823_7
3. Cao, J., Das, S.K.: Mobile Agents in Networking and Distributed Computing. Wiley, Hoboken (2012)
4. Chalopin, J., Godard, E., Métivier, Y., Ossamy, R.: Mobile agent algorithms versus message passing algorithms. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 187–201. Springer, Heidelberg (2006). doi:10.1007/11945529_14
5. Das, S., Flocchini, P., Santoro, N., Yamashita, M.: Fault-tolerant simulation of message-passing algorithms by mobile agents. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 289–303. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72951-8_23
6. Das, S., Mihalák, M., Šrámek, R., Vicari, E., Widmayer, P.: Rendezvous of mobile agents when tokens fail anytime. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 463–480. Springer, Heidelberg (2008). doi:10.1007/978-3-540-92221-6_29
7. Dieudonné, Y., Pelc, A.: Deterministic network exploration by a single agent with byzantine tokens. Inf. Process. Lett. **112**(12), 467–470 (2012)
8. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Searching for a black hole in arbitrary networks: optimal mobile agents protocols. Distrib. Comput. **19**(1), 1–18 (2006)
9. Erciyes, K.: Distributed Graph Algorithms for Computer Networks. Springer Science & Business Media, London (2013). doi:10.1007/978-1-4471-5173-9

10. Flocchini, P., Huang, M.J., Luccio, F.L.: Decontaminating chordal rings and tori using mobile agents. Int. J. Found. Comput. Sci. **18**(03), 547–563 (2007)
11. Klasing, R., Markou, E., Radzik, T., Sarracco, F.: Hardness and approximation results for black hole search in arbitrary networks. Theoret. Comput. Sci. **384**(2–3), 201–221 (2007)
12. Luccio, F., Pagli, L., Santoro, N.: Network decontamination in presence of local immunity. Int. J. Found. Comput. Sci. **18**(03), 457–474 (2007)
13. Suzuki, T., Izumi, T., Ooshita, F., Kakugawa, H., Masuzawa, T.: Move-optimal gossiping among mobile agents. Theoret. Comput. Sci. **393**(1–3), 90–101 (2008)
14. Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press, Cambridge (2000)