Fault-Tolerant Simulation of Message-Passing Algorithms by Mobile Agents

Shantanu Das* Paola Flocchini* Nicola Santoro† Masafumi Yamashita‡

Abstract

The recently established computational equivalence between the traditional message-passing model and the mobile-agents model is based on the existence of a mobile-agents algorithm that simulates the execution of message-passing algorithms. Like most existing protocols for mobile agents, this simulation protocol works correctly only if the agents are fault-free.

We consider the problem of performing the simulation of message-passing algorithms when the simulating agents may crash unexpectedly. We show how to simulate any distributed algorithm for the message-passing model in a mobile-agents system with k agents, tolerating up to $f \leq k-1$ crashes during the simulation. Two fault-tolerant simulation algorithms are presented, one for non-anonymous settings (i.e., where either the networks nodes or the agents or both have distinct identities), and one for anonymous systems (where both the network nodes and the agents are anonymous). In both cases, the simulation overhead is polynomial.

An interesting feature of the algorithm for the anonymous setting is that it allows for a *self-balancing* fault-tolerant simulation: Even though the agents may crash at any time, the algorithm ensures that the simulation proceeds flawlessly irrespective of the agent crashes and the system always stabilizes to a state where the workload is equally distributed among the remaining agents.

Finally, unlike the existing fault-free simulation algorithm, both our protocols are able to detect termination even if the simulated algorithm has no explicit termination detection.

Keywords : *mobile agents, crash failures, fault tolerance, message passing, distributed simulation, termination detection, anonymous systems*

submission to SIROCCO 2007

Corresponding author: Shantanu Das
School of Information Technology and Engineering
University of Ottawa,
Ottawa K1N 6N5
Canada.
Email: shantdas@site.uottawa.ca

*School of Information Technology and Engineering, University of Ottawa, Canada. Email:{shantdas,flocchin}@site.uottawa.ca
†School of Computer Science, Carleton University, Canada. Email: santoro@scs.carleton.ca
‡Dept. of Computer Science and Communication Engineering, Kyushu University, Japan. Email: mak@csce.kyushu-u.ac.jp

1 Introduction

1.1 The Framework

A distributed computing environment typically consists of a collection of autonomous computational entities that can communicate among each-other to perform a common task. The most common model of distributed computation is the *message-passing* model, in which the entities are connected through point-to-point links, according to some fixed topology (often represented as a graph); the entities are stationary and communicate by sending and receiving bounded sequences of bits (called messages), through the incident links (called ports).

Another model of distributed computation that has been studied recently is the *mobile agents* model. In this model the entities are mobile (rather than stationary) and their movement is constrained by the topology of the environment (which is again represented by a graph). The entities, called *agents* or *robots*, have computing and storage capabilities, and can move along the edges of the graph going from one node to an adjacent node. Each node of the network (called a host), provides a storage area (called whiteboard) for incoming agents, whose access is held in fair mutual exclusion. The communication between the agents occurs by writing notes on and reading notes from the whiteboards.

Mobile agents have been extensively studied for several years by researchers in Artificial Intelligence and in Software Engineering. They offer a simple and natural way to describe distributed settings where mobility is inherent, and an explicit and direct way to describe the entities of those settings, such as mobile code, software agents, viruses, robots, web crawlers, etc. Further, they allow to express immediately notions such as selfish behavior, negotiation, cooperation, etc. arising in the new computing environments. As a programming paradigm, the model allows a new philosophy of protocol and communication software design. As a computational universe, the model opens a variety of new challenging problems (e.g., *rendezvous*, *intruder detection*, *network decontamination*, etc.), most of them with immediate practical relevance and applicability.

In addition to the study of the new problems opened by the mobile agents model, intriguing research questions have naturally arisen on the differences (or similarities) between computing with mobile agents and computing with stationary agents. The two environments were initially compared from a systems engineering point of view by Fukuda et al.[15]. An insight on the *computational* relationship between the two models was provided by Barrière et al. [4], who noticed that any mobile agent algorithm can be simulated in the message-passing model; this immediately implies that all the impossibility results under the message-passing model hold also for the mobile-agents model. The reverse direction has been an open problem for quite a while. The question has been answered recently by Chalopin et al. [8], who proved that indeed the two models are computationally equivalent; in fact, they showed how to construct a mobile-agents algorithm Y from a message-passing algorithm X in such a way that every possible outcome of Y is a valid outcome for algorithm X. While the consequences of this theoretical result are not yet fully realized, already several results have been transferred between the two models [8]. Like most previous results on mobile agent computing, also these equivalence results and the simulation algorithm assume a *fault-free* environment.

In this paper, we consider instead systems where agents may fail by crashing at any time, and we investigate the problem of developing a *fault-tolerant* simulation. Thus the model considered in this paper is more realistic with respect to the typical networked environments where transmission errors or inconsistencies may cause some agents to be dropped or deleted from the system. We show that even in such an environment, we can simulate any message-passing algorithm, while tolerating an arbitrary number of agent crashes.

Moreover, we show that the simulation can be done with explicit *termination detection*; that is, if the simulated algorithm terminates (explicitly or not), the simulation will be able to detect this within a finite time and all (surviving) agents would reach terminal states. This is an improvement on the existing simulator of [8] that would terminate explicitly only if the simulated algorithm would do so.

1.2 Main Results

In a mobile agent system, the most common type of fault that can occur is that an agent suddenly disappears or is destroyed while moving from one node to another. This kind of fault, called *agent crash* (or simply, crash), may occur in networked environments due to various reasons, e.g. network congestion, or unavailability of a host. In this paper, we will focus on this type of faults, and assume that the system is otherwise reliable. In other words, an agent is safe once it has arrived at a node; however it may crash while moving between nodes. Let us denote by k the number of agents and by f the number of crashes in the system.

We first consider the *non-anonymous* setting; that is when the network nodes have distinct identifiers. Notice that, in this case, each agent can choose a distinct identity for itself (if it does not already have one); thus, also agents are non-anonymous. Further note that the case when the network is anonymous but the agents have distinct identities is equivalent to the above one (since agents can assign a distinct names to each node using the whiteboards).

For this setting, we design a simulation algorithm that is robust against any number of crashes, short of the collapse of the entire system. In fact, as long as at least one agent is alive, our algorithm is able to effectively simulate the message-passing computation achieving the same results as would have been obtained in the absence of any faults. We prove that, if the simulated algorithm terminates, the simulation will explicitly terminate within finite time; this will happen regardless of the number $f \leq k-1$ of crashes that occur during the simulation, and even if the simulated computation does not have termination detection. Our simulation algorithm, *DisSimulate*, has an overhead of at most $O(n)$ moves per agent, for each message transmitted in the original algorithm, where n is the number of nodes. The local memory required by each agent is $O(n \log \Delta)$, where Δ is the maximum degree in the network.

We then consider the *anonymous* setting; that is when neither the network nodes nor the agents have distinct identifiers. We show that anonymity of both networks and agents does not impede the ability of mobile agents to simulate any message-passing computation for anonymous networks. In fact, the algorithm *DisSimulate* can be executed on anonymous networks too. However, as expected, the cost is much higher in this case and depends on how efficiently the network can be traversed. In the worst case, an exponential overhead can be incurred for simulating each message transmission. To overcome this difficulty, we propose another algorithm, *AnSimulate*, that simulates any message-passing computation on an anonymous network even when $f \leq k-1$ of agents crash during the simulation. Also in this case, if the simulated algorithm terminates, the simulation will explicitly terminate within finite time; even if the simulated computation does not have an explicit termination detection mechanism. The overhead of *AnSimulate* is $O(m+n \cdot k)$ agent moves in total for each message exchanged in the simulated computation (where m is the number of edges in the graph/network). The local memory requirement is same as before.

An interesting feature of algorithm *AnSimulate* for the anonymous setting is that it allows for a *self-balancing* fault-tolerant simulation: Even though the agents may crash at any time, the algorithm ensures that the simulation proceeds flawlessly irrespective of the agent crashes and the system always stabilizes to a state where the workload is equally distributed among the remaining agents.

1.3 Related Work

Although most of the classical results on distributed computing are based on the message-passing stationary-agent model, some of these algorithms use the concept of token-passing, where a mobile object, called the *token* moves among the stationary processes in the network. In some algorithms the possession of a token gives a process the authority to act and it is generally required that at any time, exactly one of the nodes in the network holds the token. However for some algorithms like leader election [1, 18] there can multiple tokens simultaneously moving through the network and these tokens also carry information. Algorithms for network exploration (e.g. [1, 21]) are typically described in terms of a travelling process or entity (sometimes modelled as a finite automata, e.g. [14]). These algorithms can be thought of as the first "mobile-agent" algorithms, as opposed to message-passing algorithms. However in all these algorithms, a mobile agent (or travelling token or automaton) is created by a stationary process to perform a particular task (e.g. exploration) after which it returns to process that created it and is destroyed there.

On the other hand, the mobile-agent model considered in this paper, consists of fully autonomous entities that are not associated with any stationary process and these entities are continuously performing computational tasks while moving among a network of data-repository nodes called whiteboards. Certain problems which are specific to this model, and have been studied recently are *agent-rendezvous*, (e.g., [19, 28]), *intruder capture* (e.g., [3, 6]), *network decontamination* (e.g., [12, 22]), and *black hole search* (e.g., [11, 17]). For recent surveys, see [13, 20]. The problem of leader election or spanning tree construction have been studied under both the mobile-agents model (e.g., [4, 9, 10]) and the message-passing model (e.g., [16, 18]). However, there has not been many studies on the differences (or similarities) between computing with mobile agents and computing with stationary agents. Fukuda et al. [15] have compared the two systems from a systems engineering point of view. Barrière et al. [4] showed that any mobile agent algorithm can be simulated in message-passing model,

which immediately implies that all the impossibility results under the message-passing model, also hold for the mobile-agents model. Chalopin et al. [8] recently showed the simulation in the other direction. These results imply that the two models are computationally equivalent.

There have been numerous studies [2, 7, 24, 27] on computability in anonymous networks under the message-passing model, in terms of which tasks can be computed on a given network and under what conditions. Most of these results would hold under the mobile-agents model too, however the complexity of doing tasks in the mobile-agents model would be different.

The study of mobile-agents systems has mostly been limited to fault-free environments. One exception is the investigation of systems with a *black hole*, i.e. a highly harmful node that destroys any agent arriving at the node. In this case, the research has mostly focussed on the problem of locating the black hole (e.g., see [11, 17]). The issue of computing with agents that can disappear (i.e. crash) anywhere in the network has not been considered before, to the best of our knowledge.

2 Terminology and Definitions

The S.A. model: In the message-passing model, the computational entities are stationary and are connected by point-to-point communication links. We call this the *stationary-agents* (SA) model, which can be described as follows. The computing environment is modelled by a connected undirected graph $G(V, E)$. Each vertex[1] of the graph G is associated with a fixed computational entity, called a stationary agent. Each node also contains a local memory which is accessible only to the agent associated with this vertex. Each agent can perform any number of computational steps, it can read from and write to the local memory of the node and it can send messages and receive messages on each edge connected to the node. The agents are reactive entities, i.e. they react in response to external stimuli, e.g. the receipt of a message. The state of an agent is defined by the contents of the local memory. Initially some of the agents would be in the special state "Initiator"; such agents would start the computation process spontaneously. The initial value stored in the local memory at a node defines its identity. Thus, the nodes have distinct identities only if the initial contents of memory at each node is different.

The edges incident to a node are labelled by local orientation $\lambda = \{\lambda_v : v \in V\}$, where for each vertex u, $\lambda_u : \{(u, v) \in E : v \in V\} \rightarrow \{1, 2, \ldots, degree(u)\}$ defines the labelling on its incident edges. We shall use $\lambda(e)$ to denote the labels on the edge $e = (u, v)$ i.e. the pair $(\lambda_u(u, v), \lambda_v(u, v))$. The agent at node u can send a message m on any incident edge $e = (u, v)$ using the primitive SEND$(m, \lambda_u(e))$. In this case, the agent at node v receives the information $(m, \lambda_v(e))$, within a finite amount of time.

The M.A. model: The *mobile-agents* (MA) model is similar to the above model, with the following differences. As before the environment is modelled by the labelled graph (G, λ); however the nodes of the graph are not associated with any fixed computational entities. Instead, there are k computational entities (called mobile agents) each of which may be located at any of the nodes of the graph at any time during the computation. The agents can perform computations and they can move along the edges of the graph. Each agent also has its individual memory that is accessible only by that agent and is called its *notebook*; This local memory moves with the agent when it travels from one node to another. The contents of the *notebook* defines the state of the agent (in particular, it contains a special variable called *Next-Node*). Each agent starts in the same initial state with the notebook containing only the algorithm to be executed. The node where an agent starts from is called its *homebase*.

In this model, the nodes of the graph are just repositories of data, with no "intelligence" (i.e. computational ability) associated with them. The local memory at each node, called the *whiteboard* of that node, is accessible to any agent that is physically present at this node. The contents of the *whiteboard* defines the state of the node. Access to a whiteboard occurs in fair mutual-exclusion.

An agent at any node v can read and modify the contents of the whiteboard of node v as well as its own notebook memory, and it can leave node v through any incident edge $e(v, w)$. When an agent leaves a node v through an edge $e(v, w)$, the agent either reaches node w within a finite amount of time or permanently disappears (i.e. crashes). Initially the agents are all identical and indistinguishable from each other (no unique identities), and they execute the same protocol. In general, a mobile agent at any node v would repeatedly execute the following cycle of steps:

[1] We use the terms 'vertex' and 'node' interchangeably.

1. [READ] Read the contents of the whiteboard of node v to the agent's notebook.
2. [COMPUTE] Perform a sequence of computations modifying the contents of the agent's notebook.
3. [WRITE] Write to the whiteboard of node v, part of the results of the computation.
4. [MOVE] If *Next-Node*= $x > 0$, then leave the node v through the edge labelled x. Otherwise if *Next-Node*=0, remain at node v.

During the time an agent A executes steps 1 to 3 at node v, the whiteboard of node v would be exclusively accessible to agent A. (Prior to executing step-1, the agent obtains a lock on the whiteboard and it releases the lock at end of step-3.)

Assumptions: We make the following assumptions.

[A1] Each communication link (represented by an edge of the graph G) satisfies the first-in-first-out property; So, if agent B leaves node v through the edge $e(v, w)$ after agent A left through the same edge, then agent B would arrive at node w after agent A arrives there.

[A2] An agent may crash while traversing an edge (i.e. during the MOVE step) but it cannot die while performing some computation at a node (i.e. during the READ-COMPUTE-WRITE steps) .

[A3] At most $k-1$ agents may crash (i.e. $f \leq k-1$). Agent crashes are permanent; Once an agent crashes, it may not become alive again.

3 Simulation in Non-Anonymous Systems

In this section we consider the case when the network nodes are non-anonymous; that is each node of the graph G is provided with a distinct identity that is initially stored in the local memory(whiteboard) of the node. In such a network, we can assume that also the agents have distinct identities: should they not have one initially, each agent can acquire the identity of the node where initially resides (conflicts resolved by using the mutual-exclusion access to the whiteboard). Notice that the case when the network nodes are anonymous but the agents have distinct identities is equivalent to the previous ones: a unique id can be assigned to the nodes by the agents (e.g. during a collective traversal).

Consider an arbitrary (message-passing) algorithm—we shall call it algorithm Z—being executed on a network (G, λ), in the conventional stationary-agent (S.A.) model. Such an algorithm can be described as follows:

Algorithm *Z*

- *An initiator, upon starting the algorithm, executes the following steps:*

Step-I1 Initiate the algorithm and do local computation(possibly generating messages to send)

Step-I2 Send zero or more messages through some of the ports.

Step-I3 Wait for messages to arrive from one or more ports.

- *Any entity, on receiving a message m, executes the following steps:*

Step-1 Read message m and do local computation(possibly generating messages to send)

Step-2 Send zero or more messages through some of the ports.

Step-3 Wait for messages to arrive from one or more ports.

The algorithm is said to have terminated when every node is in Step-3 (passive mode) and there are no messages in transit.

To simulate such an algorithm in a mobile agent system, we need to execute the active steps (Step-1 and Step-2) at each node. This involves performing local computation at the nodes and delivering messages between nodes. The idea of the simulation is simple – the mobile agents can move from node to node delivering the

messages; An agent that delivers a message m to a node x, can also perform the local computation at node x that results from the receipt of the message m. Using the client-server paradigm, we can think of each node as a client which requires some service (transporting messages, performing local computation etc.) and the agents as mobile servers which move from node to node performing the required services.

For effectively simulating the message-passing computation using mobile agents that can fail at any time, we need to address the following issues:

[FR] Fairness: Every message that is generated at a node should be delivered within a finite time to its destination.

[TD] Termination Detection: Once all messages have been delivered and the execution of the original algorithm Z terminates, then each agent should be able to detect this and stop the simulation.

To ensure fairness, every agent would periodically visit each node to deliver the messages generated at that node. Thus, even if an agent dies, the others would keep servicing the nodes (delivering the messages). Note, however that an agent may die on the way while delivering some message m. In that case, another agent has to take over the task of delivering that particular message. So, we need to keep track of which messages have been delivered successfully. For this purpose, we maintain two message queues at each node– the *To-Be-Delivered*(TBD) message queue and the *Messages-Received*(MR) queue. Any messages generated at a node v, is added to the TBD queue at v, along with the port number $p = \lambda_v(e)$ of the edge e through which the message needs to be sent. A message can be removed from the TBD queue when it is known that it has been received successfully and written to the MR queue at the receiving end. A message in MR queue can be removed when it is read and the actions corresponding to the receipt of this message are executed. We keep such messages in a third queue called *Messages-Executed*(ME) queue. Each message m generated at a node v and to be sent through port p would be assigned an identifier, called MID(m); The TBD queue would store the original message m along with the (MID(m),p) pair as key, while the MR queue would store the message with (MID(m),q) as key, where q is the label of the port through which this message was received. The ME queue only needs to store the identifier (MID(m),q) and not the contents of the original message.

The simulation of the message-passing algorithm Z would be started at those nodes which are in the special state "*Initiator*". Recall that the original algorithm may not have an explicit termination detection mechanism. So, in order to ensure that the agents can detect when the computation has terminated, we would maintain a dynamic forest-like structure among the nodes of the graph, using an idea of Shavit and Francez (see [25]). The initiator nodes would be the root nodes and every other (active) node x would contain a link to its parent node i.e. the node y which send the first message to node x to activate it. We would maintain at each node x a child-list containing the links to all its (active) children, i.e. all those nodes which were activated by messages from node x and are still active. The leaf nodes would be those that did not send any messages or sent messages to already active nodes. Once a leaf node completes its local computation and it does not have any messages to send, then it is removed from the child-list of its parent node and becomes inactive. Once a root node has no children and no further messages to send, it becomes inactive too. If all the root nodes become inactive then the simulation is terminated.
The simulation algorithm executed by each agent A is given below:

Algorithm *DisSimulate*

I Explore the network G and construct a traversal path P_A that starts and ends at the homebase of A, and visits every other node at least once.

II Walk through path P_A and at each node v which is in the state *Initiator* do the following -

1. Initiate the algorithm Z at node v and perform all local computation steps until the first time a message needs to be sent or received.
2. Update the state of node v by writing to the whiteboard the current state of execution. In particular, set *node-state* to "Processing" if a message needs to be sent, and to "Waiting" if a message needs to be received.
3. If a message m has to be sent through port p, then add (MID(m),p) to the TBD queue.

III Walk through path P_A and at each node x do the following -

1. If the TBD queue is not empty, then execute procedure *Deliver-Message*.
2. If node-state = "Processing", then continue with the local computation at current node until a message needs to be sent or received. Update the state of node x by writing to the whiteboard.
3. While the MR queue is not empty, execute procedure *Receive-Message*.
4. If there are no messages neither in the TBD queue nor in the MR queue, the Child-List is empty, and the node-state is "waiting", then write 'TERM' (for "terminated") on the whiteboard of x. Then go to the parent node y(if any) and delete x from the Child-List at node y. Now return back to node x.

IV If all the nodes visited in the previous step had a 'TERM' symbol written on their whiteboard then terminate the algorithm. Else repeat previous step.

where the procedures *Deliver-Message* and *Receive-Message* are as follows:

Procedure *Deliver-Message*

1. Let (m_i, p) be the first entry in the TBD queue. Leave the current node x through port p to reach node y (say, through port q), where $e = (x, y)$, $\lambda_x(e) = p$ and $\lambda_y(e) = q$.
2. If the message (m_i, q) is not present in the MR queue(or, the ME queue) at node y, then do the following:
   Add the message (m_i, q) to the MR queue,
   Delete any 'TERM' symbol (if present) from the whiteboard of node y,
   If the parent-link of node y is not set, set it to q.
   Set result to *Success*.
   Else set result to *Fail*.
3. Return back to node x and delete message (m_i, p) from the TBD queue (if present).
   If the parent-link of node y was set to q, then add the link $p = \lambda_x(e)$ to the Child-list of node x.
4. If the result is *Fail* and the TBD queue is not empty, then go back to the first step.
   Otherwise, return the result.

Procedure *Receive-Message*

1. Let (m_i, q) be the first entry in the MR queue. Remove the message (m_i, q) from the MR queue and perform the local computation at the current node that results from receiving this message from port q.
2. If a message m has to be sent through port p, then add $< MID(m), p >$ to the TBD queue. Update the state of the current node by writing to the whiteboard.
3. Add (m_i, q) to the ME queue

We shall now show the above algorithm satisfies the Fairness [FR] and Termination Detection [TD] properties. (Due to the space constraint, most of the proofs in this section and the next have been omitted and can be found in the appendix.) We denote by L_A the length of the traversal path P_A constructed by in step-(I) by an agent A, and we define L to be the maximum length of such a path; i.e.,

$$L = max_A\{L_A\}$$

Lemma 3.1 *An agent starting at any node v of G can construct a path of length $O(n)$ that starts and ends at v and visits each vertex of G at least once.*

Since the nodes of the network are uniquely identifiable, each agent can construct the required path by a simple depth first traversal of the graph. Thus, $L \leq 2 \cdot n$.

Lemma 3.2 *Each message generated at some node x and added to the TBD queue at position r is delivered to its destination after at most $3 \cdot L \cdot r$ moves by any single agent.*

Due to assumption [A3], at least one agent remains alive. So, every message would be delivered within a finite amount of time. This ensures the fairness of message delivery.

Lemma 3.3 *Every message is delivered and executed exactly once and no message is repeated.*

Lemma 3.4 *When any agent A terminates the simulation, then the following conditions hold: (i) No node has any more messages to send and (ii) there are no messages in transit.*

Lemma 3.5 *Once the execution of the original algorithm Z terminates (i.e. when all messages have been delivered and executed), then every agent that is alive would terminate after at most $L \cdot n$ moves.*

Lemma 3.6 *During any execution of the algorithm DisSimulate, the size of the TBD queue at any node v would be at most* 1 *and the size of the MR queue would be at most $(k \cdot L)$. The size of ME queue at any node v would be at most the degree of the node.*

Notice that, if we can ensure that each node appears exactly once in the traversal path P_A of any agent A, then the maximum size of the MR queue would decrease to $k \cdot d$. In particular, if r is the maximum number of times a single node v appears in the traversal path of some agent A, then the size of the MR queue is never more than $k \cdot d \cdot r$.

Theorem 3.1 *The result obtained (i.e. the final state of the nodes) in any possible execution of algorithm* DisSimulate*, would be exactly identical to the result of some possible execution of the original algorithm Z in the S.A. model.*

Notice that the above theorem holds, irrespective of the number of agents failing or crashing, if at least one agent is alive. So, if there exists a S.A. algorithm for solving any computational problem in a network (G, λ), then the same problem can be solved in the mobile agent system with just one agent. In other words, we can say the following:

Observation 3.1 *A mobile agent system with at least one agent in a network of n nodes, is computationally as powerful as a stationary agent system with n agents.*

Let us now measure the cost of the simulation.

Theorem 3.2 *During the simulation, the total number of moves made by the agents is at most $O(k \cdot L)$ per message exchanged in an execution of the original algorithm Z.*

Let us analyze the memory overhead required for our simulation algorithm. Notice that each agent needs to store, in its local memory, a copy of the traversal path P_A. Thus the amount of additional memory required by each agent for the simulation is $O(L \log \Delta)$, where Δ is the maximum degree of the graph.

The amount of additional memory required at the nodes (other than what is required for storing the state of the algorithm Z) needs to be enough for storing the message queues.

4 Simulation in Anonymous Systems

In the previous section, we investigated systems having distinct identifiers for either the nodes of the network or the agents, or both. In this section, we consider the simulation of message-passing algorithms in anonymous networks by anonymous agents.

4.1 Employing the Existing Solution

The same simulation algorithm of the previous section can possibly be executed on anonymous networks too. Notice that even if the graph is anonymous, it is still possible for an agent to find a traversal path in the graph that visits every node, provided that the agent knows the size n of the graph or at least an upper bound on n.

Lemma 4.1 *An agent A starting at a node v of an anonymous graph G, can construct a path P(represented by a sequence of edge labels) of finite length that starts and ends at v and is guaranteed to visit each node at least once.*

Proof : Let V_i be the set of all nodes which can be reached from v by a path (not necessarily simple) of length i. (Note that $V_0 = \{v\}$.) The agent A simply traverses each outgoing edge from V_i to construct V_{i+1}. Once the agent has constructed V_{n-1}, it would have traversed every path of length less than n starting from its homebase and thus every node of the graph would have been visited (This follows from the fact that the graph is connected). This gives us the required traversal path. ■

Notice that the traversal path constructed in the proof above is of length $O(\Delta^n)$. However in most cases, it is possible to construct much shorter traversal paths. In particular, if the agent can map the graph, then

it can always construct a traversal path of length $O(n)$ (i.e. linear in the size of the graph). As the efficiency of our simulation algorithm depends on the length of the traversal path used, we would like the length of the traversal path to be as small as possible, preferably linear (or at most quadratic) in the size of the graph, n. The problem of finding such a path is in fact, related to the problem of constructing the map of a graph and this is very difficult to solve, in the presence of multiple identical agents. One approach (see [9]) is to elect a leader among the agents and then allow the leader to label the graph, so that it becomes possible to map the graph. We do not want to use such an approach here, because in that case our algorithm may fail when the leader agent crashes before completing its job. We want each agent to independently traverse the graph using its own traversal path. This can be achieved for example, by employing the method of *random walk of a graph*. However, the discussion of random walks would be outside the scope of the current paper.

When using deterministic methods, the problem of finding a linear length traversal path of a graph, can be solved in certain cases, (e.g. if the *symmetricity* [26] of the graph is equal to one). In general however, it is not always possible to find a traversal path of linear (or even polynomial) length, visiting all the nodes of an arbitrary anonymous graph.

Observation 4.1 *There exists graphs (G, λ) with k agents placed among the nodes of G, such that the agents having only the knowledge of $n = |G|$ and executing identical deterministic algorithms, cannot obtain a path of length polynomial in n that is guaranteed to visit every node of G.*

Observation 4.2 *The algorithm* DisSimulate *when executed on anonymous networks, has an exponential overhead, in terms of agent moves, for delivering each message.*

4.2 In Absence Of Good Traversal Paths

When it is not possible to construct efficient traversal paths, we can use a different approach for simulating a message-passing computation on a graph G. We partition the graph G among the k agents and each agent would be responsible for the subgraph of G that it owns (we call this the territory of the agent). However, as some of the agents may die, the task of a dead agent must be taken over by some other (alive) agent. So, in our algorithm, each agent simulates the computation within its territory, while periodically checking if any of its neighboring agents are dead and annexing the territory of any dead agent. Initially, the agents mark their territories in the graph G (using procedure EXPLORE), thus partitioning G into k disjoint trees. In the algorithm, when we say that an agent marks a node v, we mean that the agent writes a particular symbol on the whiteboard of node v to denote the fact that this node has been visited. Similarly, an edge $e = (u, v)$ of G is marked by writing on the whiteboard of the adjacent nodes u and v.

The algorithm given below, simulates a given message-passing algorithm Z, on the graph G, using k mobile agents. Initially, each agent knows the steps of the algorithm Z and each initiator node is marked as such. The following fact will be used in the algorithm:

Fact 1 *Given any starting node v of G, every edge $e \in G$ can be uniquely identified by using the sequence of edge-labels for the path from v to e. We use the notation $\boldsymbol{id}_v(e)$ to denote such an identifier for an edge e at node v.*

ALGORITHM *AnSimulate*:

Phase 0: Each agent A executes procedure EXPLORE to obtain its territory T_A. The territory T_A is a tree rooted at the homebase, and all the other nodes contain a link marked as home-link which connects this node to its parent in the tree T_A. Let n_A be the size of T_A. Agent A then traverses its territory T_A and at each node v that it visits — if v in the state "initiator", then agent A initiates the algorithm Z at node v performing all local computation steps until the first time a message needs to be sent or received; Agent A then updates the state of node v and if any message m is to be sent through the port p, then the pair (m, p) is added to the *To-Be-Delivered*(TBD) queue at node v.

Phase $i \geq 1$: Agent A, (if alive) executes the following steps:

STEP 1: Agent A does a depth-first traversal of its territory T_A. During the traversal, for each node u that it visits, agent A does the following:

- If the TBD queue of node u is not empty, then execute procedure *Deliver-Message*;
- If node-state = "Processing", then continue with the local computation at current node until a message needs to be sent or received. Update the state of node x by writing to the whiteboard.
- While the MR queue is not empty, execute procedure *Receive-Message*.
- If there are no messages in both the TBD queue and MR queue, the Child-List is empty, and the node-state is "waiting", then write 'TERM' on the whiteboard of u. Then go to the parent node v(if any) and delete u from the Child-List at node v. Now return back to node u.
- Write DONE(i, n_A) on the whiteboard of u.

If during Step-1, agent A finds an ANNEXED$(j, n_B$,**id**$(e'))$ mark in its homebase, then agent A goes to the edge e' and marks this edge as Tree-edge (For the next phase, $n_A \leftarrow n_A + n_B$ and $T_A \leftarrow T_A + \{e'\} + T_B$.) In this case, agent A skips STEP-2 and jumps to STEP-3 to update its territory.

STEP 2: Agent A starts a depth-first traversal of its territory. During the traversal, for each external edge $e = (u, v)$ incident to some node u in its territory, it traverses the edge e to reach the other end v, reads DONE(j, n_B) from whiteboard[2] at v and takes the following actions:

- If $(j < i)$ or, $(j = i$ AND $n_B < n_A)$, then go to the root-node x of the tree T_B containing v and write ANNEXED$(i, n_A$,**id**$_x(e))$ (only if there is no other ANNEXED mark at node x).
- If successful in writing the ANNEXED mark, then return to e and mark this edge as a Tree-edge.(For the next phase $n_A \leftarrow n_A + n_B$ and $T_A \leftarrow T_A + \{e\} + T_B$.)

STEP 3: Agent A updates its territory T_A to include all territories that it annexed and those annexed by the agents that it defeated. If agent A itself was defeated, then it adds the territory of the agent C that defeated it and all territories annexed by C. The home-links of the nodes in the territory are updated and the value of n_A is modified accordingly. If all nodes in its territory had 'TERM' written on the whiteboards then agent A executes procedure *Termination-Detection* to check if the termination condition has been reached and if so, stops. Otherwise agent A goes to phase $i + 1$.

Procedure *Termination-Detection*

For $r = 1$ to k, do
    If there is some node in T_A which does not have a 'TERM' mark on its whiteboard, then return false;
    Else write 'TERM(r)' on the whiteboard of every node in T_A.
    For each non-tree edge $e = (u, v)$ incident to a node $u \in T_A$,
        Traverse edge e to reach node $v \in T_B$ (say),
        If node v has no 'TERM' mark, then return false;
        Else if found a 'TERM(j)' mark and $j < r$ then,
            go to the root of tree T_B and mark it with 'TERM(r)' (if not already marked so)
            If successful, merge T_A with T_B, using the edge e and continue;
If $r = k$, then return true;

Procedure *EXPLORE*

1. Set *Path* to empty; Mark the homebase as explored and include it in territory T;
2. While there are unexplored edges at the current node u,
    select an unexplored edge e,
    mark link $l_u(e)$ as explored and then traverse e to reach node v;
    If v is already marked (or v contains another agent),
      mark e as a non-tree edge;
      return back to u;
    Otherwise
        mark node v as explored and mark link $l_v(e)$ as home-link;
        Add link $l_v(e)$ to *Path*;
        Add the edge e and node v to the territory T;

[2]If no such mark is found, it reads DONE$(j = 0, n_B = 0)$.

3. When there are no more unexplored edges at the current node,
   If Path is not empty then,
      remove the last link from Path, traverse that link and repeat Step 2;
   Otherwise, Stop and return the territory T;

The procedures *Deliver-Message* and *Receive-Message* are same as before.

In the above algorithm, an agent A annexes the territory of another agent B, if either (i) B has died (or B is slower than A) or, (ii) if during some phase i, B has a smaller territory than A. After agent A annexes the territory of agent B, these two territories are merged and both the agents(if alive) continue the simulation in the bigger territory. The algorithm ensures that the territory of an agent is always a tree. This ensures that each agent completes a single traversal (i.e. a single phase) in $O(n)$ moves.

We give below the proof of correctness of our algorithm and analyze its complexity. Due to assumption [A3], there are some agents which survive till the end of the algorithm—we shall call such agents "alive agents". The lemma below proves fairness of message delivery.

Lemma 4.2 *During algorithm AnSimulate, the following always holds:*
1. *The edges marked as tree-edges form a spanning forest of G, containing at most k trees (each rooted at some homebase).*
2. *Every node is visited by an alive agent at least once in every k phases.*
3. *An alive agent completes each phase within a finite amount of time.*
4. *Whenever an alive agent visits a node v, the top-most message in the TBD queue of v is delivered to its destination (unless the queue is empty).*
5. *Every message generated at a node v is delivered within a finite amount of time.*

Lemma 4.3 *When an agent A completes procedure* Termination-Detection *with a return value of true, then every message corresponding to the execution of algorithm Z, has been delivered and there are no messages in transit.*

Lemma 4.4 *When every message corresponding to the execution of algorithm Z, has been delivered, every alive agent terminates within at most k phases.*

Theorem 4.1 *Algorithm* AnSimulate *correctly simulates any given massage-passing algorithm Z, even if up to $f \leq k-1$ agents crash.*

Let us now analyze the cost of the simulation.

Lemma 4.5 *During every phase in which no agents crash, at least one message is delivered unless there are no more pending messages.*

Lemma 4.6 *In each phase of algorithm AnSimulate, the number of moves made by the agents in total is $O(m \cdot k)$.*

The above result follows from the fact that each edge is traversed a constant number (at most six) times by any single agent. However, we can make the following modification to the algorithm to reduce its communication complexity. Whenever an agent traverses an external edge during step-2 of a phase, it can mark the edge such that no other agent follows it and traverses the same edge again in this phase. Thus any non-tree edge would be traversed by at most one agent from each side. However, a tree edge may be traversed by all agents that share that territory. So, the number of moves per phase would be $O(m + n \cdot k')$ for k' alive agents. Thus, we have the following result, due to Lemma 4.5.

Theorem 4.2 *For the (modified) algorithm AnSimulate, the overhead for delivering each message is $O(m + n \cdot k')$, where k' is the number of surviving agents.*

We would like to remark that the large overhead for message delivery is due to the fact that we have to deal with failures of any number of agents at any time during the simulation. In environments without agent failures, it is always possible to simulate a message-passing computation much more efficiently. For example, the algorithm given by Chalopin et al.[8] for the fault-free environment requires $O(n)$ agent moves per message delivery in the worst case.

5 Conclusions and Open Problems

We proposed and studied methods for simulating any message-passing computation in a mobile agent system with faulty agents. In particular, we have shown how to simulate a given message-passing algorithm, in a mobile agent system, while tolerating the crash-fault of any number of agents, provided that at least one agent is alive. Thus, agent crashes do not restrict the computational power of a mobile agent system. Another interesting observation is that a mobile agent system of n nodes with at least one agent is computationally as powerful as a stationary agent system with n agents.

We presented an algorithm *DisSimulate*, for simulating a message-passing computation in a labelled network (or, a network which can be explored efficiently). In this algorithm, the agents work independently of one another, with no communication among the agents and this makes this algorithm very robust. However, one problem is sometimes the workload is not evenly distributed among the agents, i.e. in the worst case, all the messages may be getting delivered by a single agent, even though the other agents may still be alive.

For anonymous networks or networks where node identities are not visible to the agents, we gave another algorithm *AnSimulate*, where the network is partitioned into disjoint parts which are serviced by different groups of agents. It is interesting to note that this algorithm has the self-stabilizing property. Even though the agents may crash at any time, the algorithm ensures that the simulation proceeds flawlessly irrespective of the agent crashes and the system always stabilizes to a state where the workload is equally distributed among the remaining agents.

In the present paper, we have only considered agent crash faults, but not node crashes. In the mobile agent systems considered here, each node is just a data repository; So a node crash in such a system implies a whiteboard crash i.e. all data stored in the corresponding whiteboard would be deleted. Such faults can be dealt with using the known fault tolerance techniques for the message-passing (S.A.) model. In particular, given any algorithm for the S.A. model that is t-crash tolerant, the same algorithm can be simulated in a mobile agent system (using our proposed method), to tolerate up to t crash faults of the nodes, irrespective of the number of agents failing.

One of the limitations of our results is the assumption that agents can only fail while traversing an edge. We have not considered the possibility of an agent failing while performing computation at a node, because in such a case, the whiteboard of the node may remain locked and thus inaccessible to all other agents, which can create a deadlock. Future studies on this problem should be directed towards finding a way to deal with crashes of agents inside a node, while avoiding the deadlock situation.

References

[1] Y. Afek and E. Gafni. Distributed algorithms for unidirectional networks. *SIAM Journal on Computing*, 23(6):1152–1178, 1994.

[2] D. Angluin, "Local and global properties in networks of processors", In *Proc. 12th ACM Symp. on Theory of Computing (STOC '80)*, 82–93, 1980.

[3] L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Capture of an intruder by mobile agents. In *Proc. 14th ACM Symp. on Parallel Algorithms and Architectures (SPAA'02)*, pages 324–332, 2002.

[4] L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Can we elect if we cannot compare? In *Proc. 15th ACM Symp. on Parallel Algorithms and Architectures (SPAA'03)*, pages 200–209, 2003.

[5] M. Bender, A. Fernandez, D. Ron, A. Sahai, and S. Vadhan. The power of a pebble: Exploring and mapping directed graphs. In *Proc. 30th ACM Symp. on Theory of Computing (STOC'98)*, pages 269–287, 1998.

[6] L. Blin, P. Fraigniaud, N. Nisse and S. Vial. Distributed chasing of network intruders. In *Proc. 13th Colloquium on Structural Information and Communication Complexity (SIROCCO'06)*, pages 70–84, 2006.

[7] P. Boldi and S. Vigna. An effective characterization of computability in anonymous networks. In *Proc. of 15th Int. Conference on Distributed Computing (DISC'01)*, pages 33–47, 2001.

[8] J. Chalopin, E. Godard, Y. Métivier and R. Ossamy, Mobile agents algorithms versus message passing algorithms, In *Proc. 10th Int. Conf. on Principles of Distributed Systems (OPODIS'06)*, pages 187-201, 2006.

[9] S. Das, P. Flocchini, A. Nayak, and N. Santoro. Distributed exploration of an unknown graph, In *Proc. 12th Coll. on Structural Information and Communication Complexity (SIROCCO'05)*, pages 99-114, 2005.

[10] S. Das, P. Flocchini, A. Nayak, and N. Santoro. Effective elections for anonymous mobile agents, In *Proc. 17th International Symposium on Algorithms and Computation (ISAAC'06)*, pages 732-743, 2006.

[11] S. Dobrev, P. Flocchini, G. Prencipe and N. Santoro. Finding a black hole in an arbitrary network: optimal mobile agents protocols. *Distributed Computing*, to appear. Preliminary version in *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC'02)*, pages 153–162, 2002.

[12] P. Flocchini, F.L. Luccio and M. Huang. Decontamination of chordal rings and tori using mobile agents *International Journal of Foundation of Computer Science*, to appear, 2007.

[13] P. Flocchini and N. Santoro. Distributed security algorithms by mobile agents In *Proc. 8th International Conference on Distributed Computing and Networking (ICDCN'06)*, 2006.

[14] P. Fraigniaud and D. Ilcinkas. Digraph exploration with little memory. In *21st Symp. on Theoretical Aspects of Computer Science (STACS'04)*, pages 246–257, 2004.

[15] M. Fukuda, L.F. Bic, M.B. Dillencourt, and F. Merchant. Messages versus messengers in distributed programming. In *Proc. 17th international Conference on Distributed Computing Systems (ICDCS '97)*, pages 347–354, 1997.

[16] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

[17] R. Klasing, E. Markou, T. Radzik and F. Sarracco. Hardness and approximation results for black hole search in arbitrary networks. *Theoretical Computer Science*, to appear, 2007.

[18] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient distributed leader finding algorithms. *ACM Trans. Program. Lang. Syst.*, 12(1):84–101, 1990.

[19] E. Kranakis, D. Krizanc, and E. Markou. Mobile agent rendezvous in a synchronous torus. In *Proc. 7th Latin American Theoretical Informatics Symposium (LATIN' 06)*, pages 653-664, 2006.

[20] E. Kranakis, D. Krizanc, and S. Rajsbaum. Mobile agent rendezvous: A Survey. In *Proc. 17th international Conference on Distributed Computing Systems (SIROCCO '06)*, pages 1–9, 2006.

[21] S. Kutten. Stepwise construction of an efficient distributed traversing algorithm for general strongly connected directed networks or: Traversing one way streets with no map. In *Proc. of 9th Int. Conference on Computer Communication (ICCC'88)*, pages 446–452, 1988.

[22] F. Luccio, L. Pagli and N. Santoro, Network decontamination in presence of local immunity *International Journal of Foundation of Computer Science*, to appear, 2007.

[23] N. Norris. Universal covers of graphs: Isomorphism to depth N-1 Implies isomorphism to all depths. *Discrete Applied Mathematics*: 56(1), pages 61–74, 1995.

[24] N. Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *Proc. of the 18th annual ACM Symp. on Principles of Distributed Computing (PODC '99)*, pages 173–179, 1999.

[25] N. Shavit and N. Francez. A new approach to detection of locally indicative stability. In *Proceedings of 13th International Colloquium on Automata, Languages and Programming, (ICALP'86)*, pages 344–358, 1986.

[26] M. Yamashita and T. Kameda. Computing on an anonymous network. Technical Report LCCR 87-15, Simon Fraser University, Vancouver, 1987.

[27] M. Yamashita and T. Kameda. Computing on anonymous networks: Part I–Characterizing the solvable cases. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):69–89, 1996.

[28] X. Yu and M. Yung. Agent rendezvous: A dynamic symmetry-breaking problem. In *Int. Coll. on Automata Languages and Programming (ICALP'96)*, pages 610–621, 1996.

APPENDIX

Proof of Lemma 3.2

Proof : To deliver a message successfully an (alive) agent A makes two moves (to and back). So, during a single traversal of the path P_A, if agent A finds some message in the queue at each node it visits, then it would make $3 \cdot L$ moves. If node x appears only once in the path P_A, then after r such traversals, the message at position r would be delivered. So, as long as there is an alive agent A, it would succeed in delivering that message after $3 \cdot L \cdot r$ moves. (Note that the other $k-1$ agents may have died, without helping agent A in its task.) ■

Proof of Lemma 3.3

Proof : Consider a message m that has to be sent from node u to node v on the edge $e(u, v)$, where $\lambda(e) = (p, q)$. When the message (m, q) is delivered at node v, it is written on MR queue. Due to the mutual exclusion property of the whiteboards, only one agent can write the message (m, q) to the MR queue. If any other agent reads the message (m, p) from the TBD queue of node u and attempts to deliver it, the agent would find the message (m, q) in either the MR queue or the ME queue of node v. When *Receive-Message*(m, q) is executed by an agent at a node v, the agent also deletes the message (m, q) from the MR queue of node v. An agent would obtain access to the whiteboard at the start of procedure *Receive-Message*, it would relinquish the whiteboard only after the message (m, q) has been deleted from the MR queue of node v. Hence every message is delivered and executed exactly once. ■

Proof of Lemma 3.4

Proof : We define a node to be in *active* state if either there are some messages in its TBD queue or MR queue, or if its child-list is not empty. When a node u is marked with 'TERM', then none of the above holds and the node is said to be *inactive*. Any node v, other than the initiator nodes, becomes active only on receiving a message from another node u and in that case it becomes the child of node u. Thus, every active node is either an initiator (i.e. a root node in the spanning forest) or belongs to the tree rooted at some initiator. Note that once an initiator node becomes inactive, then it can never become a root node again. So, if an agent A on traversing the graph finds all nodes (including the initiators) are inactive, then no node can ever become active again. However, if any of the conditions of the lemma is false then there must be at least one active node. Hence the lemma holds. ■

Proof of Lemma 3.5

Proof : Consider the instance when the last message (in the given execution) is received and executed at a node v. At this instance, all TBD and MR queues at every node would be empty and each node would be in state "waiting". During the next L moves of any agent A (at least one such agent exists due to assumption [A3]), all the nodes which do not have any child-links(i.e. *leaf* nodes) would have 'TERM' written on their whiteboards. In every subsequent L moves, the nodes at the next higher level(i.e. the new *leaf* nodes) would have 'TERM' written on their whiteboards. So, after at most $L \cdot n$ moves by agent A, every node would have 'TERM' written on its whiteboard and thus the agent would terminate. ■

Proof of Lemma 3.6

Proof : An agent can add at most one message to the TBD queue at node v during each visit to node v. Whenever any agent A adds a message to the TBD queue, either the TBD queue was empty before this message was added or, agent A had succeeded in removing one message from the TBD queue in the previous step. This means that the contribution from each agent is one, when the queue is empty and it is zero in all other cases. So, the size of the TBD queue can be at most 1.

Each time an agent A visits node v it removes all pending messages from the MR queue of that node. Between two consecutive visits to a node v, the agent may make at most L visits to other nodes and if each

such visit results in a message being sent to node v, then the total contribution of a single agent, to the MR queue of a node v, would be at most L. Hence the result follows.

Whenever a message received from port p is added to the ME queue of node v, it can replace the previous message (if any) that was received from the same port. This is possible because whenever a message m_i is received on edge e, it implies that the previous message m_{i-1} send on the same edge, has already been deleted from the TBD queue at the source. The FIFO property of the channels (assumption [A1]) ensures this. ■

Proof of Theorem 3.1

Proof : By Lemmas 3.2-3.5 it follows that algorithm *DisSimulate* successfully and fairly delivers and executes every message corresponding to algorithm Z. For any execution of *DisSimulate* on the network (G, λ), consider the timing sequence in which messages are generated and received by the algorithm. Notice that such a sequence of operations satisfies the causal order of sending and receiving messages. Now, consider the execution of algorithm Z in an equivalent S.A. system (G, λ). Since the system is asynchronous, messages may be sent and received in any possible order (satisfying the causal relationship). In particular, one of these possible ordering of the send and receive operations would correspond to the order in which the messages are generated and delivered during the execution of algorithm *DisSimulate*. So, this particular execution of algorithm Z would be equivalent to the execution of algorithm *DisSimulate*, i.e. the contents of each node would be exactly same during these two executions and thus, the same result would be obtained. ■

Proof of Theorem 3.2

Proof : As a consequence of Lemma 3.2, r messages are delivered by an agent, using $3 \cdot r \cdot L$ agent moves. Now, in the worst case, only one agent (say agent A) may be delivering all the messages while the other agents just keep following agent A, (always arriving at each node just after agent A). Thus, in this case, each agent would make $O(L)$ moves in delivering a message and the total agent moves would be $O(k \cdot L)$ per message. ■

Proof of Lemma 4.2

Proof : (1) This clearly holds at the end of procedure EXPLORE. Whenever two trees are merged this property is maintained.

(2) A node v that is in the territory of an agent A, would be visited once in every phase, unless agent A dies. If agent A dies during phase i, node v would still be visited by its neighboring agent B in phase $i+1$, unless agent B dies during phase $i+1$, in which case, its neighbor agent C would visit node v during phase $i+2$, and so on. Since $f < k$ agents may crash, so node v would be visited at least once in k phases. More precisely, node v is visited at least $r - f$ times during r phases of the algorithm.

(3) This follows from the fact that there is no waiting during any phase of algorithm AnSimulate.

(4) This follows from the definition of procedure *Deliver-Message*.

(5) This follows from parts (2),(3),(4) above and the fact there is always at least one alive agent (assumption [A3]). ■

Proof of Lemma 4.3

Proof : If procedure Termination-Detection returns true, then every node in agent A's territory has TERM(k) written on it. This implies that nodes in neighboring territories are at least marked with TERM($k-1$), while territories at a distance of two are marked with $(k-2)$ and so on. Since the network is partitioned into at most k territories, every node in the network would have a TERM mark, which implies that no node has any more messages to send and there are no messages in transit. ■

Proof of Lemma 4.4

Proof : Once all messages have been delivered, every node would become inactive. When an alive agent A visits such a node, it would mark it with 'TERM'. So all the nodes in agent A's territory would be marked with 'TERM' and the agent would start procedure Termination-Detection. As there are no nodes in G, which does

not have a TERM mark, the procedure would return true after at most k phases and agent A would terminate. ■

Proof of Theorem 4.1

Proof : The fairness of message delivery is ensured by Lemma 4.2, while the lemmas **??** ensure that algorithm *AnSimulate* terminates correctly. As before, the procedures *Deliver-Message* and *Receive-Message* ensure that every message is delivered and executed exactly once and further, the delivery of messages follows the causal ordering of receiving and sending messages. Thus, any execution of algorithm *AnSimulate* would correspond to some possible execution of algorithm Z in the S.A. model. ■

Proof of Lemma 4.5

Proof : Notice that every vertex v is visited once in every phase i, unless some agent crashed in phase i. Whenever node v is visited, the first message (if any) in the TBD queue of node v would be delivered. Thus, except for those phases (at most f) during which some agents crash, every phase results in the delivery of at least one message. ■