Distributed algorithms for dynamic survivability of multiagent systems[⋆]

V.S. Subrahmanian[1], Sarit Kraus[2], and Yingqian Zhang[3]

[1] Dept. of Computer Science, University of Maryland, College Park, MD 20742
vs@cs.umd.edu
[2] Dept. of Computer Science, Bar-Ilan University, Ramat Gan, 52900 Israel
sarit@cs.biu.ac.il
[3] Dept. of Computer Science, University of Manchester, M13 9PL, UK
zhangy@cs.man.ac.uk

Abstract. Though multiagent systems (MASs) are being increasingly used, few methods exist to ensure survivability of MASs. All existing methods suffer from two flaws. First, a centralized survivability algorithm (CSA) ensures survivability of the MAS - unfortunately, if the node on which the CSA exists goes down, the survivability of the MAS is questionable. Second, no mechanism exists to change how the MAS is deployed when external factors trigger a re-evaluation of the survivability of the MAS. In this paper, we present three algorithms to address these two important problems. Our algorithms can be built on top of *any* CSA. Our algorithms are completely distributed and can handle external triggers to compute a new deployment. We report on experiments assessing the efficiency of these algorithms.

1 Introduction

Though multiagent systems are rapidly growing in importance, there has been little work to date on ensuring the survivability of multiagent systems (MASs for short). As more and more MASs are deployed in applications ranging from auctions for critical commodities like electricity to monitoring of nuclear plants and computer networks, there is a growing need to ensure that these MASs are robust and resilient in the face of network outages and server down times.

To date, there has been relatively little work on survivability of MASs. Most approaches to ensuring survivability of MASs are based on the idea of replicating or cloning agents so that if a node hosting that agent goes down, a copy of the agent residing on another network location will still be functioning. This paper falls within this category of work. However, existing replication based approaches suffer from two major flaws.

The first major flaw is that the survivability algorithms themselves are *centralized*. In other words, even though the agents in the MAS may themselves be distributed across

[⋆] Sarit Kraus is also affiliated with University of Maryland. This work was supported in part by the Army Research Lab under contract DAAD19-03-2-0026, the CTA on Advanced Decision Architectures, by ARO contract DAAD190010484, by NSF grants 0205489, IIS0329851 and IIS0222914.

the network, the survivability algorithm itself resides on a single node. Thus, if the node hosting the survivability algorithm goes down along with all nodes containing some agent in the MAS, then the system is compromised. This way of "attacking" the MAS can be easily accomplished by a competent hacker.

The second major flaw is that the survivability algorithms do not *adapt* to changes that affect the survivability of the MAS. Most algorithms assume that once the survivability algorithm tells us where to replicate agents, we just replicate the agents at the appropriate nodes and then ignore survivability issues altogether. It is clearly desirable to continuously (or at least regularly) monitor how well the MAS is "surviving" and to respond to changes in this quantity by redeploying the agent replicas to appropriate new locations.

We present three *distributed* algorithms to ensure that a multiagent system will survive with maximal probability. These algorithms *extend centralized algorithms for survivability* such as those developed by [1] but are completely distributed and are adaptive in the sense that they can dynamically adapt to changes in the probability with which nodes will survive. The algorithms are shown to achieve a new deployment that preserves whatever properties the centralized algorithm has. For example, in a recent paper on survivability, Kraus et. al. [1] develop a centralized algorithm called COD for computing deployments of MASs that maximize the probability of survival of the deployment. If our distributed algorithms were built on top of COD, the resulting deployments created by our system would also maximize probability of survival.

We have also developed a prototype implementation of our algorithms and conducted detailed experiments to assess how good the algorithms are from three points of view: (i) what is the CPU time taken to find a deployment, (ii) what is the amount of network time used up in redeploying agents, and (iii) what is the survivability of deployments.

2 Assumptions

Throughout this paper, we assume that an *agent* is a program that provides one or more services. Our framework for survivability is independent of the specific agent programming language used to program the agent. In addition, we assume that a *multiagent application* is a finite set of agents. We will develop the concept of a *deployment agent* introduced in the next section that can be used by the MAS to ensure its own survivability.

We assume that a network is a fully connected graph $G = (V, E)$, i.e. $E = V \times V$. In addition, we assume that each node $n \in V$ has some memory, denoted $space(n)$, that it makes available for hosting agents in a given multiagent system. We also use $space(a)$ to denote the space requirements of an agent a. If A is a set of agents, $space(A)$ is used to denote $\sum_{a \in A} space(a)$.

When a set D of nodes in a network $G = (V, E)$ *goes down*, the resulting network is the graph $G' = (V - D, E - \{(v_1, v_2) \mid (v_1, v_2) \in E$ and either $v_1 \in D$ or $v_2 \in D\})$.

A *deployment* of a MAS $\{a_1, \dots, a_n\}$ w.r.t. a network $G = (V, E)$ is a mapping $\mu : V \to 2^{MAS}$ such that for all $1 \leq i \leq n$, there exists a $v \in V$ such that $a_i \in \mu(v)$.

Suppose μ is a deployment of $\{a_1, \ldots, a_n\}$ w.r.t. a network G and suppose $G' = (V', E')$ is the resulting network when some set D of nodes goes down. μ *survives the loss of* D iff the restriction μ' of μ to $V - D$ is a deployment w.r.t. G'. Intuitively, this definition merely says that a MAS survives when a set of nodes goes down if and only if at least one copy of each agent in the MAS is still present on at least one node that did not go down. We demonstrate our problem using the following example.

Example 1. Suppose $V = \{n_1, n_2, n_3, n_4\}$ and $MAS = \{a, b, c, d\}$. A current deployment is given by: $\mu_{old}(n_1) = \{a\}, \mu_{old}(n_2) = \{b, d\}, \mu_{old}(n_3) = \{a, b\}, \mu_{old}(n_4) = \{b, c\}$.

Suppose the system administrator of node n_4 announces that this node will go down in an hour in order to perform an urgent maintenance task. It is easy to see that μ_{old} will not survive this event as agent c will not be present in any of the nodes. Thus, μ_{old} should be changed: a copy of agent c should be deployed on one of the nodes $n_1, ..., n_3$ and additional copy of b may also be deployed in $n_1, ..., n_3$. Space restrictions of these nodes may lead to additional changes in the deployment, e.g., a copy of d may be moved from node n_2 to node n_3.

3 Distributed Multiagent Survivability

In this section, we provide three alternative algorithms to ensure the survivability of a MAS. As mentioned above, we assume that our distributed survivability algorithms build on top of some arbitrary, but fixed centralized survivability algorithm CSA. Several such algorithms exist such as those in [1]. We now describe each of these three algorithms. **Note that all copies of the deployment agent perform the actions here, not just one copy** (if only one copy performed the computations, then we would just have a centralized algorithm).

As mentioned earlier, our algorithms will use a *deployment agent* (*da* for short) which will ensure survivability. *da* is added to the MAS as an additional survivability agent.

Definition 1. *Suppose MAS is a multi-agent application. The* survivability enhancement *of* MAS*, denoted* MAS^**, is the set* $MAS \cup \{\text{da}\}$ *where da is a special agent called* deployment agent*.*

The rest of this section focuses on different ways of designing and implementing the deployment agent *da*.

3.1 The ASA1 Algorithm

The ASA1 algorithm deploys a copy of *da* in every node of the network. We make the following assumptions about *da*: (i) *da* knows the current deployment, (ii) whenever a new deployment needs to be computed, *da* is triggered, (iii) *da* is built on top of any arbitrary centralized survivability algorithm CSA.

As *da* is located in each node, we will assume that for any $n \in V$, $space(n)$ is the available memory on n excluding the space for *da*.

Whenever the *da* agents are notified that a new deployment needs to be computed, each copy of the *da* agent performs the following steps:

1. It examines the current deployment μ_{old};
2. Once *da* is told to redeploy by an external process, it uses the CSA algorithm to compute a new deployment μ_{new};
3. *da* stores the difference between μ_{old} and μ_{new} in a special data structure called a *difference table*. The difference table dif has the following schema:
   - Node ($string$): node id for all $n \in V$;
   - Deploy (set of $string$): agents' current deployments μ_{old};
   - Insrt (set of $string$): agents that are presently not located in the node but need to be allocated according to the new deployment μ_{new};
   - Remv (set of $string$): agents that are presently located in the node but need to be deleted from it according to the new deployment μ_{new};
4. Each copy of *da* at each node looks at its *Insrt* and *Remv* columns and makes a decision on how to delete and/or add agents from its node.

Notice that at any given instance in time, all the deployment agents on all nodes have the same difference table. Our key task is to design step 4. Before doing this, we present an example of a difference table.

Example 2. Consider the MAS and μ_{old} of Example 1. Consider a new deployment: $\mu_{new}(n_1) = \{a, b\}$, $\mu_{new}(n_2) = \{b, c\}$, $\mu_{new}(n_3) = \{a, d\}$, and $\mu_{new}(n_4) = \{d\}$. In this case, the difference table between μ_{old} and μ_{new} is given by Table 1.

Node	Insrt	Remv	Deploy
n_1	b		a
n_2	c	d	b, d
n_3	d	b	a, b
n_4	d	b, c	b, c

Table 1. A difference table generated by *deployment agent*

Adding and/or deleting agents to/from nodes can be performed according to the difference table. However, these operations should be handled very carefully as there are two constraints that must be satisfied during the whole re-deployment process:

- *space*: while these operations are being performed, the space constraint on each node must be satisfied;
- *copies of agents*: at any point in time during step (4), there must exist at least one copy for each agent $a \in MAS$ in the network.

Example 3. To see why Step 4 is complex, consider the difference table in Table 1. One may be tempted to say that we can implement the insertions and deletions as follows: (i) Insert c on n_2. (ii) Delete d from n_2. Notice however that we can insert c on n_2 only if there is enough space on n_2 to accommodate b, c, d simultaneously (as otherwise the host node n_2 may reject the insertion of c) for space violations. Alternatively, one may be tempted to first delete d from node n_2 to free space to insert c - but this means that agent d has disappeared from all nodes and is hence lost for ever !

Before presenting our algorithm for deleting/adding agents, we first present a few definitions of concepts that will be used in the algorithm.

Definition 2. *An agent a can be safely deleted from node n (denoted by $safeDel(a,n)$) if the number of copies of agent a in the* Deploy *column of the difference table is larger than the number of copies of agent a in the* Remv *column.*

When an agent can be *safely deleted*, we are guaranteed that at least one copy of the agent is present elsewhere on the network. In our running example (Table 1), the only agent that can be safely deleted is agent b at node n_3.

We use $Insrt(n)$, $Remv(n)$ and $Deploy(n)$ to denote the insert list, the remove list and the deploy list of node $n \in V$ in the difference table. The implementation of *da* in ASA1 algorithm is based on a set of logical rules governing the operations of *da*. We first present these rules before describing the algorithm in detail. The rules use the following action predicates (predicates representing actions are used in much the same way as in Kowalski's and Green's formulations of planning, cf. [2]).

- $ADD(a,n)$: Add agent $a \in MAS$ to node $n \in V$;
- $DEL(A,n)$: Delete a set of agents $A \subseteq MAS$ from node $n \in V$;
- $SWITCH(A,n,A',n')$: Switch two sets of agents $A \subseteq MAS$ and $A' \subseteq MAS$ that are located on nodes n and n' respectively;
- $\mathsf{remdif(A,L,n)}$ and $\mathsf{insdif(A,L,n)}$: Suppose A is a set of agents, L is a string in $\{Remv, Insrt, Deploy\}$, and n is a node. $\mathsf{remdif(A,L,n)}$ removes all nodes in the L-list of node n in the difference table. Likewise, $\mathsf{insdif(A,L,n)}$ inserts all nodes in A into the L list of node n's entry in the difference table.

Note that $Insrt(n)$ represents the $Insrt$ field of node n in the difference table. It specifies what new agents must be inserted into node n. In contrast, $\mathsf{insdif(A,Insrt,n)}$ specifies that $Insrt(n)$ must be updated to $Insrt(n) \cup A$, i.e. it refers to an update of the difference table itself. In the example of Table 1, $\mathsf{remdif(\{b\},Deploy,n_2)}$ causes the deploy field associated with n_2 to be reset to just $\{d\}$ instead of $\{b,d\}$.

We now introduce the rules governing the execution of these actions.

Rule 1 *The first rule says that if A is a set of agents each of which can be safely deleted from node n, then A can be removed from node n.*
$DEL(A,n) \leftarrow (\forall a \in A) safeDel(a,n)$

Rule 2 *This rule says that if a set A of agents is deleted from node n, we need to update the difference table by removing A from the remove and deploy lists of node n.*
$\mathsf{remdif(A,Remv,n)} \wedge \mathsf{remdif(A,Deploy,n)} \leftarrow DEL(A,n)$

Rule 3 *This rule says that an agent a can be added to node n if there is sufficient space on node n to accommodate a's memory needs.*
$ADD(a,n) \leftarrow (space(n) - space(Deploy(n)) \geq space(a)$

Rule 4 *If agent a is added to node n, we must remove its id from the insert column and add it to the deploy column of node n.*
$\mathsf{remdif(\{a\},Insrt,n)} \wedge \mathsf{insdif(\{a\},Deploy(n))} \leftarrow ADD(a,n)$

Rule 5 *These rules says that two sets of agents, A deployed on node n and A' on node n', can be switched if: A' is a subset of the insert set on node n as well as A' is in the deleted set of node n'; A is a subset of the remove set on node n and it is also in the added list of node n'; furthermore, the space constraints on switching A and A' between n and n' must be satisfied.*

$SWITCH(A, n, A', n') \leftarrow$
$A' \subseteq Remv(n') \wedge A' \subseteq Insrt(n) \wedge A \subseteq Remv(n) \wedge A \subseteq Insrt(n') \wedge$
$CHKSWITCH(A, n, A', n')$.
$CHKSWITCH(A, n, A', n') \leftarrow$
$(Space(n) - space(Deploy(n)) + space(A) \geq space(A')) \wedge$
$(space(n') - space(Deploy(n')) + space(A') \geq space(A))$.

$SWITCH(A, n, A', n')$ performs appropriate ADD and DEL actions on agents at the appropriate nodes.
$(\forall a \in A')ADD(a, n) \wedge (\forall a \in A)ADD(a, n') \wedge DEL(A', n') \wedge DEL(A, n)$
$\leftarrow SWITCH(A, n, A', n')$

Rule 6 *This rule says when $SWITCH(A, n, A', n')$ is performed, we must update the difference table.*
$\mathsf{remdif(A, Remv, n)} \wedge \mathsf{remdif(A', Remv, n')} \wedge \mathsf{remdif(A', Insrt, n)}$
$\wedge \mathsf{remdif(A, Insrt, n')} \wedge \mathsf{remdif(A, Deploy, n)} \wedge \mathsf{insdif(A', Deploy, n)}$
$\wedge \mathsf{remdif(A', Deploy, n')} \wedge \mathsf{insdif(A, Deploy, n')}$
$\leftarrow SWITCH(A, n, A', n')$.

Rule 7 *The rules below deal with the case where there is no agent that can be* safely deleted *from node n (the case shown in rule 1) and there is no current available space for adding an agent (as described in rule 3) and there is no direct switch that could be performed (the case of rule 5). That is, when more than two nodes are involved with switch, we need the following rules.*
$SWITCH(A, n, A', n') \leftarrow$
$A' \subseteq Remv(n') \wedge A \subseteq Remv(n) \wedge (\exists B \subseteq A')B \subseteq Insrt(n) \wedge$
$CHKSWITCH(A, n, A', n')$.
$(\forall a \in A')ADD(a, n) \wedge (\forall a \in A)ADD(a, n') \wedge DEL(A', n') \wedge DEL(A, n)$
$\leftarrow SWITCH(A, n, A', n')$

When switching A and A', if we move an agent b to a node where b is not the desired agent in the new deployment, we should delete b from that node in the future process, that is, we should add b to the delete list of the node.
$\mathsf{remdif(A, Remv, n)} \wedge \mathsf{remdif(A', Remv, n')} \wedge \mathsf{remdif(A, Deploy, n)} \wedge$
$\mathsf{insdif(A', Deploy, n)} \wedge \mathsf{remdif(A', Deploy, n')} \wedge \mathsf{insdif(A, Deploy, n')}$
$\leftarrow SWITCH(A, n, A', n')$
$\mathsf{insdif(\{b\}, Remv, n)} \leftarrow (\forall b \in A')b \notin Insrt(n) \wedge SWITCH(A, n, A', n')$.
$\mathsf{remdif(\{b\}, Insrt, n)} \leftarrow (\forall b \in A')b \in Insrt(n) \wedge SWITCH(A, n, A', n')$.
$\mathsf{insdif(\{b\}, Remv, n')} \leftarrow (\forall b \in A)b \notin Insrt(n') \wedge SWITCH(A, n, A', n')$.
$\mathsf{remdif(\{b\}, Insrt, n')} \leftarrow (\forall b \in A)b \in Insrt(n') \wedge SWITCH(A, n, A', n')$.

Our algorithm to redeploy a MAS is based on the above set of rules.

Algorithm 1 ASA1*(Ne, MAS, dif)*

```
(⋆ Input: (1) network $Ne = (V, E)$            ⋆ )
(⋆        (2) multiagent application $MAS$ ⋆)
(⋆        (3) current difference table $dif$  ⋆ )

1. $flag_1 = true$
2. while $flag_1$ do (* changes are needed by diff table *)
     – if (for all $n \in V$, $Remv(n) = \emptyset$ and $Insrt(n) = \emptyset$), then $flag_1 = false$;
     – else, do
        (1) $flag_2 = true$, $flag_3 = true$
        (2) while $flag_2$, do (* do updates; diff table updated *)
             (a) $flag_2 = false$
             (b) for each $n \in V$, do
                  A. $A = Remv(n)$ (* do all possible deletions *)
                  B. if $A \neq \emptyset$, then
                     $(dif, flag_2) = DEL(A, n, dif, flag_2)$
             (c) for each $n \in V$ do
                  A. $A = Insrt(n)$
                  B. if $A \neq \emptyset$, then (* add all agents you can *)
                     $(dif, flag_2) = ADD(A, MAS, n, dif, flag_2)$
        (3) for each $n \in V$, do
            if $flag_3$, then
              i. $A = Insrt(n)$
             ii. if $A \neq \emptyset$, then (* switch agents that could not be added before *)
                 $(dif, flag_3) = SWITCH(A, MAS, n, dif, flag_3)$
```

The function DEL($A, n, dif, flag$) receives as input: (1) a set of agents A, (2) a node n (3) a current difference table dif and (4) a flag. For each agent in A, the algorithm checks if it can safely delete the agent; if so it deletes the agent from n and updates the dif table. It returns the updated dif table and sets the flag to be true if any agent was deleted. The function ADD(A, MAS, n, dif) receives as an input (1) a set of agents A, (2) a multiagent application MAS, (3) a node n, (4) the current difference table dif, and (5) a flag. For each agent $a \in A$ if there is enough space on n to deploy a, i.e., $space(n) - space(Deploy(n)) \geq space(a)$, it adds a to n, updates the dif table and changes $flag$ to indicate an agent has been added to n. It returns: (1) the dif table, and (2) the flag.

The $SWITCH$ function uses a subroutine called $CHKSWITCH(A, n, A', n', dif, MAS)$. This function checks to see if any space overflows occur when exchanging a set A of agents current on node n with a set of agents A' currently on node n'. If no space overflow occurs, it returns true - otherwise it returns false.

Algorithm 2 *SWITCH($R, MAS, n, dif, flag$)*

```
(⋆ Input:  (1) a set of agents $R$                 ⋆)
(⋆         (2) multiagent application $MAS$ ⋆)
(⋆         (3) node id $n$                          ⋆)
(⋆         (4) current difference table $dif$   ⋆)
(⋆         (5) $flag$                               ⋆)
(⋆ Output (1) updated difference table $dif$   ⋆)
(⋆         (2) $flag$                               ⋆)
```

```
1. for each agent $a \in R$, if $(flag)$, do
   if there exists a set $A \subseteq Remv(n)$, $n' \in V$ and a set $A' \subseteq Remv(n')$ such that
     a. $a \in A'$, and
     b. $CHKSWITCH(A, n, A', n', dif, MAS) = true$
     then
     a. switch $A$ and $A'$ between nodes $n$ and $n'$
     b. $Remv(n) = Remv(n) \setminus A$, $Remv(n') = Remv(n') \setminus A'$,
        $Deploy(n) = Deploy(n) \cup A' \setminus A$,
        $Deploy(n') = Deploy(n') \cup A \setminus A'$
     c. for each $b \in A'$, do
        if $b \notin Insrt(n)$ then $Remv(n) = Remv(n) \cup \{b\}$
        else $Insrt(n) = Insrt(n) \setminus \{b\}$
     d. for each $b \in A$, do
        if $b \notin Insrt(n')$ then $Remv(n') = Remv(n') \cup \{b\}$
        else $Insrt(n') = Insrt(n') \setminus \{b\}$
     e. update $dif$
     f. $flag = false$
2. return $dif$ and $flag$
```

The following lemmas are needed to prove that ASA1is correct.

Lemma 1. *Each execution of action DEL, ADD, and $SWITCH$ always results in a decrease on the number of agents in column $Remv$ or $Insrt$.*

Proof. Rules 2 and 4 clearly show that the actions $DEL(A, n)$ and $ADD(a, n)$ remove agents from $Remv(n)$ and $Insrt(n)$ respectively. Now consider the $SWITCH$ action. When switching agents between two nodes only (Rule 5), $SWITCH(A, n, A', n')$ removes A from $Remv(n)$, A from $Insrt(n')$, A' from $Remv(n')$, and A' from $Insrt(n)$, as shown in Rule 6. In the case of Rule 7, where more than two nodes are involved in the switch, action $SWITCH(A, n, A', n')$ adds at most $A+A'-1$ agents in $Remv(n)$ and $Remv(n')$, while removing at least $A + A' + 1$ agents from $Remv$ and $Insrt$ of node n and n'. This shows that performing each action must reduce the number of agents in the $Remv$ or $Insrt$ columns. □

Lemma 2. *In each iteration of the while loop shown in Step 2 of algorithm* ASA1*, at least one action (DEL, ADD, or $SWITCH$) must be executed.*

Proof. In Step (b) of the while loop (2), all possible deletions DEL will be performed if agents in $Remv$ can be safely deleted according to Rule 1. In the loop of Step (c), all possible additions ADD will be done based on constraints shown in Rule 3. Even if no action is performed in the while loop (2), in Step (3), according to Rule 5 and Rule 7, there must exist two sets of agents on two different nodes such that action $SWITCH(A, n, A', n')$ can be performed. This shows that for each while loop in Step 2, at least one action on agents will be executed. □

Theorem 3 (Correctness). *Suppose the rules (Rule 1 - 7) are applied according to the order listed. Then the sequence of actions performed by Algorithm* ASA1 *is the one performed by the rules.* ASA1 *always terminates.*

Proof. In ASA1, the execution of actions is determined by the rules. With the assumption that the actions of the rules are taken according to their order, actions executed by the algorithm are those entailed by the rules.

In Algorithm ASA1, the while loop of Step (2) makes sure that no more agents can be safely deleted and no more agents can be added to the nodes, i.e. $flag_2 = false$. Thus, the loop in Step (2) terminates after some iterations.

For each execution of the while loop in Step 2, according to Lemma 2, there must execute at least one action on some agent at some nodes. Moreover each action must reduce the size of $Remv(n)$ or $Insrt(n)$ as explained in Lemma 1. Thus the size of $Remv$ and $Insrt$ decreases monotonically with each iteration of the while loop. Therefore the algorithm must reach a step where for all n in the network, $Remv(n) = \emptyset$ and $Insrt(n) = \emptyset$, which make $flag_1$ false, and the algorithm terminates. □

Example 4. Consider the network and the deployment of example 1. Suppose each node in the network can store a deployment agent da and two regular agents. Suppose that da were triggered and suppose they computed a new deployment as specified in example 2. Then, each copy of da computes the dif table as listed in Table 1. According to algorithm ASA1, b is first deleted from node n_3 by da located on that node and b and c are deleted from node n_4 by its deployment agent. d is not deleted in the first round because it is not safe to delete it at that stage. b is then inserted into node n_1 (copied from n_2) and d is inserted into node n_3 and n_4. d is then removed from n_2, and finally c is inserted into node n_2.

3.2 The ASA2 Algorithm

In this algorithm the deployment agent da is not located at each node. Instead, we add da to a multiagent system MAS to get an updated multiagent system MAS^* and apply the centralized algorithm on MAS^*. This returns a new deployment μ_{new} which is then executed by the deployment agent. In this case, the programming of da is somewhat different from the programming of it in ASA1 because there is no guarantee that every node has a copy of da.

Algorithm ASA2 assumes that each agent has a mobility capability, i.e., it can obtain a movement instruction from a da and perform it. In addition, each agent can delete itself. In addition, all agents in MAS as well as da satisfy the condition that whenever it receives a message from *any* da to move to another location, it does so. After performing the move, it sends a message to all deployment agents saying it has moved.

Once μ_{new} is computed by CSA, each copy of da executes an algorithm called $DELETECOPY$ that deletes all but one copy of all agents in MAS. All copies of da send messages to the agent copies to be deleted telling them to delete themselves. da copies create a plan to move and/or copy the one remaining copy of each agent to the nodes specified by μ_{new}. Note that all copies of da perform the same actions at the same time.

Algorithm 4 ASA2*($Ne, MAS^*, \mu_{old}, \mu_{new}$)*
```
(⋆ Input: (1) network $Ne = (V, E)$            ⋆)
(⋆        (2) multiagent application $MAS^*$   ⋆)
(⋆        (3) current deployment $\mu_{old}$    ⋆)
(⋆        (4) new deployment $\mu_{new}$        ⋆)
```

```
1. for each $a \in MAS^*$, do DELETECOPY($a, \mu_{old}$);
2. $flag = true$;
3. while $flag$, do
   – if (for all $n \in V$, $\mu_{old}(n) = \mu_{new}(n)$), then $flag = false$
   – else, do
     (a) $flag2 = false$, $flag3 = true$
     (b) for all $n \in V$, do
         i. $A = \mu_{new}(n)$
         ii. $flag2 = ADDCOPY(A, \mu_{old}, \mu_{new}, n)$
     (c) if ($flag2 = false$), then
           for each $n \in V$, do
             if flag3, then
         i. $A = \mu_{old}(n)$
         ii. flag3=SWITCHCOPY($A, \mu_{old}, \mu_{new}$)
```

The above algorithm uses $DELETECOPY$ (not specified explicitly due to space constraints) which uses a deterministic algorithm to delete all but one copy of each agent (e.g. via a lexicographic order on nodes). It is important that all copies of da use the same $DELETECOPY$ algorithm so that they all agree on what nodes each agent should be deleted from. Likewise $ADDCOPY(a, \mu_{old}, \mu_{new}, n)$ adds a copy of agent a to node n if there is space on node n and if the new deployment μ_{new} requires a to be in n - it does this by asking a node currently hosting a to clone and move such a copy of n. Details of $ADDCOPY$ are suppressed due to space constraints. All these algorithms update μ_{old}.

Algorithm 5 *SWITCHCOPY(A, μ_{old}, μ_{new})*

```
(⋆  Input: (1) a set agents $A$           ⋆ )
(⋆         (2) old deployment $\mu_{old}$  ⋆)
(⋆         (3) new deployment $\mu_{new}$  ⋆)
(⋆ Output: (1) flag                        ⋆ )

1. for each agent $a \in A$, do
       if there exists a set $A'$ on $n'$ such that
     (a) $a \in \mu_{old}(n')$, and
     (b) $CHKSWITCH(A, n, A', n') = true$
     then
     (a) switch $A$ and $A'$ between nodes $n$ and $n'$ and update $\mu_{old}$;
     (b) $flag = false$;
2. return $flag$
```

Example 5. Suppose nodes n_1 and n_3 of the network of Example 1 can store a da agent and two other agents. Suppose n_2 and n_4 can store only two regular agents. First, agents a,b and da are removed from node n_3. Then, agent b is removed from node n_4. The deployment agent da in node n_1 is responsible for all these deletions and for further updates. It also updates μ_{old} accordingly. b is then added to node n_1, and d and da are added to nodes n_3 and n_4. Only then is d deleted from n_2. c is then added to n_2 and then deleted from n_4.

3.3 The ASA3 Algorithm

Just as in algorithm ASA2, the deployment agent used in Algorithm ASA3 is not located on each node. Instead it is treated just like any other agent and deployed using the CSA. However, the procedure to decide on the order of deletion and adding copies of agents to nodes is that of algorithm ASA1. The behavior of the deployment agent is as follows.

> Originally, it is deployed (along with other agents) using the CSA algorithm. When survivability of one or more nodes changes, each da computes the difference table (as in the ASA1). Each da then sends a message to all agents that can be safely deleted (including, possibly a deployment agent da) telling them to delete themselves and send a message just when they are about to finish the operation. After this, they send "move" or "exchange" messages to agents one at a time. When they get an acknowledgment that the move has been performed, they send a move message to the next agent, and so on until they are done. Note that while in Algorithm ASA1, agents can be moved/copied to other nodes simultaneously, in algorithm ASA3 this is done sequentially.

The correctness proof of ASA3 is similar to the one done for ASA1. The details of the proof are omitted in the paper due to space constraints.

4 Implementation and Experimental Results

We developed a prototype implementation of all the above algorithms in Java and tested them out on a Linux PC. We used a sample of 31 existing agents to determine a distribution of agent sizes (in the 0 to 250 KB range). We ran experiments with varying network bandwidths - for space reasons we only report on experiments where the bandwidth was 100 KB/s (this is twice the bandwidth of a dial-in model, but much smaller than the bandwidth of broadband connections that may exceed 100 MB/s). The centralized survivability algorithm we used for our experiments was COD [1].

Figure 1 shows the effect of problem size on the CPU time required by the algorithms as well as the network time required to move the agents around. We used various measures of "problem size" in our experiments (such as sum of numbers of agents and nodes, ratio of number of agents to the number of nodes, etc.). Only the first is reported in figure 1 due to space constraints. The markings such as $n : 5, a : 4$ refer to a MAS of 4 agents deployed over 5 nodes. We made the following observations:

1. CPU Time: ASA1 and ASA3 always outperform ASA2 w.r.t CPU time. ASA1 and ASA3 are more or less incomparable.
2. Network Time: Again, ASA1 and ASA3 always outperform ASA2. As the problem size gets larger, ASA1 outperforms ASA3.

Due to space constraints, we are unable to present full details of all our experiments. However, the above experiments imply that ASA1 is preferable to both ASA2 and ASA3 as far as time is concerned.

Figure 2 reports some results on survivability of deployments using the three algorithms described in this paper. In the first set of experiments, we fix the size of da agent but vary the problem size, while in the second set of experiments, we change the ratio of da size to the average agent's size. As shown in Figure 2, when the problem size increases or the size of the da agent increases, the survivability of the deployments identified by ASA3 becomes higher than the survivability of deployments identified by ASA1 (note ASA2 has the same survivability as ASA3). The results demonstrate the effect of da agents on survivability of deployment. Compared with ASA2 and ASA3, ASA1 deploy more da agents in the network, and hence, the amount of available space for adding regular agents is decreased.

5 Related Work and Conclusions

To our knowledge, there are no distributed probabilistic models of survivability of a MAS. In addition, there are no works we are aware of that allow for redeployment of agents when there are changes that trigger the need to examine if a redeployment is needed. [7, 6] use agent-cloning and agent-merging techniques to mitigate agent overloading and promote system load balancing. Fan [8] proposes a BDI mechanism to formally model agent cloning to balance agent workload. Fedoruk and Deters [12] propose transparent agent replication technique. Though an agent is represented by multiple copies, this is an internal detail hidden from other agents. Several other frameworks also support this kind of agent fault tolerance. Mishra and Huang [17, 13] present a Dependable Mobile Agent System (DaAgent), which includes three protocols for recovering node and communication failures. Marin et al. [10] develop a framework to design reliable distributed applications. They use simulations to assess migration and replication costs. Kumar et al. [11] apply the replication technique to the broker agents who may be inaccessible due to system failures. They use the theory of teamwork to specify robust brokered architectures that can recover from broker failure. Our algorithms ASA1, ASA2 and ASA3 can be built on top of any of these centralized agent survivability models. The RECoMa system in [3] uses multiple servers to support matching agents to computer. Our framework assumes that any agent can be deployed on any computer, and focuses on dynamically deploying agents to increase system survivability taking into account space constraints on nodes.

Klein et al. [18] propose a domain independent approach to handling of exceptions in agent systems. This service can be viewed as a "coordination doctor", who predefines several typical abnormal situations that may arise in the system. Based on that, they monitor agent's behaviors, diagnose problematic situations and take recovery actions. Exception handling in their method is carried out by a set of collaborative agents, however, the approach itself is essentially centralized. Kaminka et al [19] utilize social knowledge, i.e. relationships and interactions among agents, to monitor the behavior of team members and detect the coordination failures. Their work focuses on exceptions concerning the agents themselves.

The fault-tolerance research area has used the N-Version Problem (NVP) approach for fault tolerance. NVP involves the "independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification" [4]. In this approach, the reliability of a software system is increased by developing several versions of special modules and incorporating them into a fault-tolerant system [14]. However, no distributed architecture for ensuring the survivability of the program ensuring survivability is discussed.

References

1. S. Kraus, V.S. Subrahmanian and N. Cihan Tacs: Probabilistically Survivable MASs. Proc. of IJCAI-03. (2003) 789-795
2. N. J. Nilsson: Artificial Intelligence: A New Synthesis. Morgan Kaufmann Publishers. San Mateo, CA, USA. 1998
3. J. A. Giampapa, O. H. Juarez-Espinosa and K. P. Sycara: Configuration management for multi-agent systems. Proc. of AGENTS-01. (2001) 230–231
4. M. Lyu and Y. He: Improving the N-Version Programming Process Through the Evolution of a Design Paradigm. IEEE Trans. Reliability. 42(2), (1993) 179-189
5. T. H. Cormen, C. E. Leiserson and R. L. Rivest: Introduction to Algorithms. MIT Press. 1990. Cambridge, MA
6. O. Shehory, K. P. Sycara, P. Chalasani and S. Jha: Increasing Resource Utilization and Task Performance by Agent Cloning. Proc. of ATAL-98.(1998) 413-426
7. K. S. Decker, K. Sycara and M. Williamson: Cloning in Intelligent, Adaptive Information Agents. In: C. Zhang and D. Lukose (eds.): Multi-Agent Systems: Methodologies and Applications. Springer-Verlag. (1997) 63-75
8. X. Fan: On splitting and Cloning Agents. Turku Center for Computer Science, Tech. Reports 407. 2001
9. D. B. Shmoys, E. Tardos and K. Aardal: Approximation algorithms for facility location problems. Proc. of STOC-97. (1997) 265–274
10. O. Marin, P. Sens, J. Briot, and Z. Guessoum: Towards Adaptive Fault Tolerance for Distributed Multi-Agent Systems. Proc. of ERSADS. (2001) 195-201
11. S. Kumar, P.R. Cohen, and H.J. Levesque: The adaptive agent architecture: achieving fault-tolerance using persistent broker teams. Proc. of ICMAS. (2002) 159-166
12. A. Fedoruk, R. Deters: Improving fault-tolerance by replicating agents. Proceedings AAMAS-02, Bologna, Italy, (2002) 737–744
13. S. Mishra: Agent Fault Tolerance Using Group Communication. Proc. of PDPTA-01, NV. 2001
14. W. J. Gutjahr: Reliability Optimization of Redundant Software with Correlate Failures. The 9th Int. Symp. on Software Reliability Engineering, 1998
15. S. Pleisch and A. Schiper: FATOMAS - A Fault-Tolerant Mobile Agent System Based on the Agent-Dependent Approach. Proc. of the DSN-01, IEEE Computer Society, (2001) 215-224
16. C. Basile: Active replication of multithreaded applications. CRHC-02-01, Univ. of Illinois at Urbana-Champaign, 2002
17. S. Mishra, Y. Huang: Fault Tolerance in Agent-Based Computing Systems. Proc. of the 13th ISCA, 2000
18. M. Klein and C. Dallarocas: Exception handling in agent systems. Proceedings of the Third International Conference on Autonomous Agents (Agents'99), (1999) 62-68
19. G. A. Kaminka and M. Tambe: Robust agent teams via socially-attentive monitoring. Journal of Artificial Intelligence Research, 12:105–147, 2000

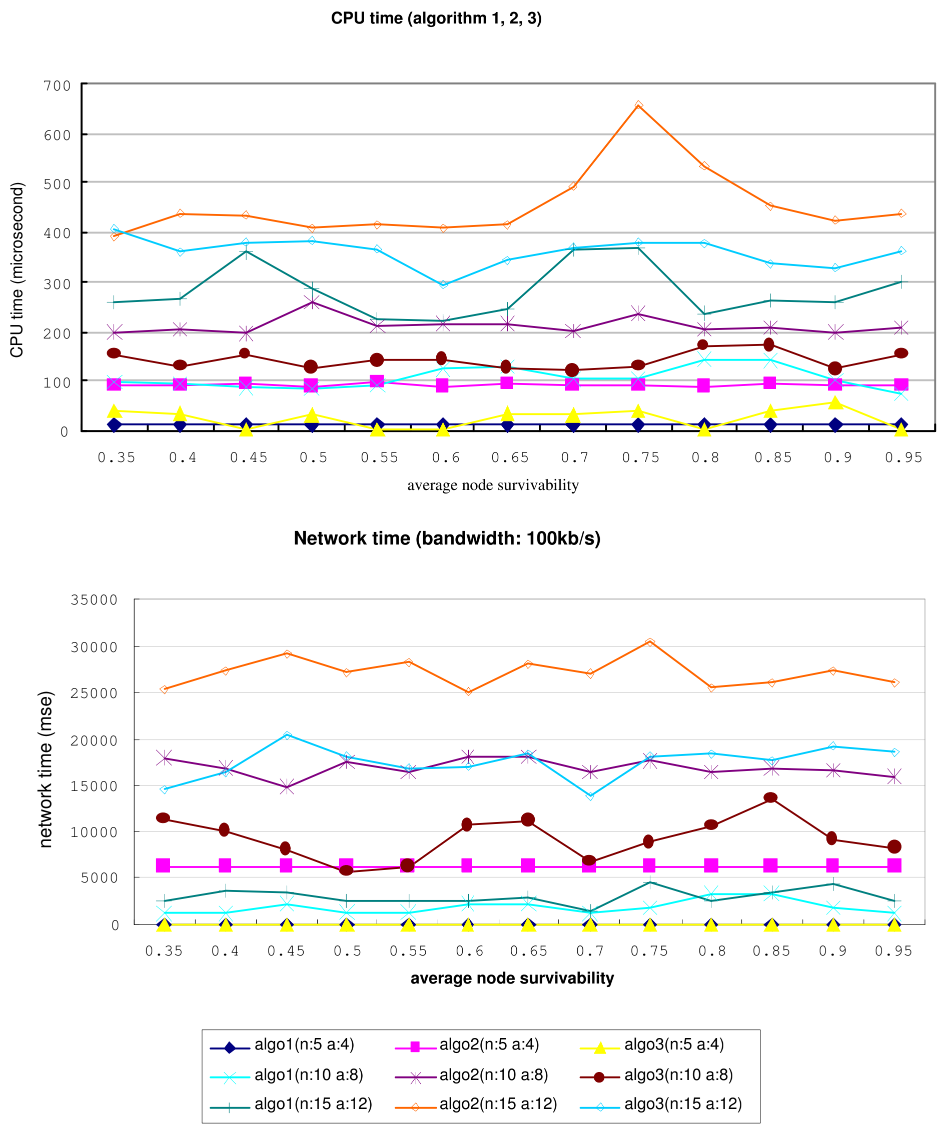
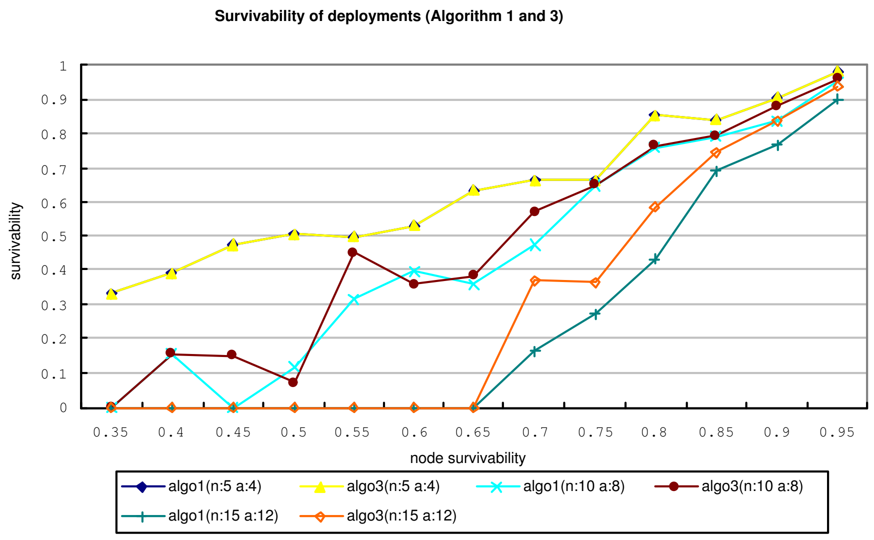


Fig. 1. CPU and Network Times of the Three Algorithms



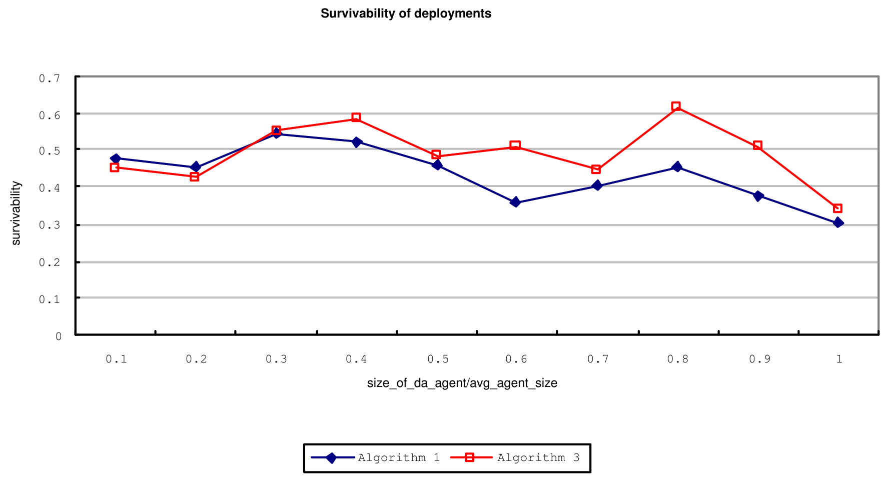


Fig. 2. Survivability of deployments by ASA1and ASA3when the problem size increases and when the size of da increases.