Exploring Robustness in the Context of Organizational Self-design

Sachin Kamboj and Keith S. Decker

Department of Computer and Information Sciences
101 Smith Hall
University of Delaware
Newark, DE 19716
{kamboj,decker}@cis.udel.edu

Abstract. Robustness is the ability of a multiagent system to recover from failures and exceptions. In particular, the system should be able to recover from task and agent failures and the failure of any single agent or group of agents should allow the graceful, predictable degradation of performance. Hence, designing robust systems is a critical challenge facing many multiagent designers.

Organizational Self-Design (OSD) has been proposed as an approach to constructing suitable organizations at runtime in which the agents are responsible for constructing their own organizational structures. OSD has also been shown to be especially suited for environments that are dynamic and semi-dynamic. However, the problem of making these self-designed organizations robust is still an open research problem that has not been studied to any considerable extent. In this paper, we focus on developing and evaluating robustness mechanisms that can be used by the agents in conjunction with OSD.

1 Introduction

Robustness may be defined as the ability of a multiagent system to recover from failures and exceptions. An exception may be defined as a departure from an "ideal" system behavior [1]. Recovery would then involve the execution of some corrective measures to reinstate the ideal system behavior.

Achieving robustness is particularly challenging in dynamic and semi-dynamic environments, since the problem characteristics, available resources or agent capabilities may change over time. Multiagent organizations for such environments must include two components — the first component is responsible for monitoring the performance of the organization and for discerning whether or not the measured performance falls within the design parameters of the organization. The second component is responsible for explicitly changing the organization if it fails to meet its design goals.

This monitoring and design may be done by an entity external to the organization (i.e. by the multiagent designer) or by the constituent agents themselves. In an extreme form of the latter approach, the agents come up with a new, implicit, one-off organization for each new problem instance. This is what happens in the contract net protocol (CNP) [2].

This latter approach is referred to in the literature as organizational self-design (OSD) [3,4] and in this approach the agents are responsible for designing their own

J.F. Hubner et al. (Eds.): COIN 2008, LNAI 5428, pp. 80–95, 2009.
© Springer-Verlag Berlin Heidelberg 2009

organizational structure at run-time. Any OSD approach needs to provide answers to questions like how many agents are needed, how are the tasks and resources divided amongst the agents and what coordination mechanisms are suitable for the problem at hand. OSD is especially suited in situations where the environment is semi-dynamic as the agents can adapt to changes in the task structures and environmental conditions, while still being able to generate relatively stable organizational structures that exploit the common characteristics across problem instances.

One application of OSD is to allocate resources in grid/cloud/volunteer computing systems. Volunteer computing [5,6] is a form of distributed computing in which a group of volunteers donate their computing resources to a cause, such as folding proteins, predicting climate change, etc. In order to do this, the volunteers have to download a client, which then connects to one or more centralized servers and requests jobs that make use of the volunteer's computing resources. The jobs are then executed on the volunteers' machines and the results are sent back to the servers. The centralized servers, in turn, need to figure out a scheduling policy that tries to perform an optimal allocation of jobs to the clients (for some definition of optimality).

The clients running on the volunteer machines can be thought of as agents. This leads to a direct mapping from the problem of determining a suitable scheduling policy for the clients to the problem of determining a suitable organization for the agents. Hence, the solution to the organizational issues, such as the allocation of agents to the subtasks of the problem being solved and the coordination of inter-agent activities, will generate a scheduling policy that can be used to allocate jobs to the agents.

One of the consequences of using volunteer computing is that the availability of any volunteer client (i.e. agent) cannot be guaranteed as (a) volunteers are free to leave at any time without any warning or penalty; and (b) the volunteers' computers might crash or be switched off. We define the non-availability of a volunteer client as an *agent failure*. Hence, if OSD is to be used to allocate tasks and resources in a volunteer computing environment, it should be able to survive agent failures.

In this paper we would like to study various approaches that can be used to increase the robustness of organizations generated through the use of OSD. Furthermore, we primarily focus on agent failures[1] in worth-oriented domains. Our approach is geared towards real-world applications in grid/volunteer/cloud computing where a large pool of agents is available though the availability of any single agent cannot be guaranteed.

A primary challenge to incorporating robustness in the OSD process is the continuously changing set of agents and the roles that they enact. This leads to a continuously changing (and distributed) organizational knowledge that must be preserved across agent failures. Furthermore, one of the principal reasons for using OSD is the amortization of organizational costs across problem instances. This amortization might come at a price — such generated organizations might be particularly susceptible to agent failure. This is because the constituent agents are responsible for enacting particular

[1] Another aspect to robustness in multiagent systems is task failures. We won't be concerned with task failures because — (a) agent failure is significantly harder than task failure because the failure of an agent results in a loss of both its organizational knowledge and its contextual problem-solving state; and (b) our approach can easily handle task failures through rescheduling[7].

goals within the organization; and the failure of a single agent, performing a critical role, could bring down the whole organization. Hence, such organizations might be *less* robust than the one-off organizational schemes (such as CNP).

We feel that robustness has not been studied in the context of OSD to any significant extent. Whereas we [4] alluded to this problem in our previous paper, we did not present any algorithms and did not discuss the problem in any depth. The algorithm presented in [8] would respond to a single agent failure by performing a complete reorganization — an extremely expensive process. [9] also address robustness by reorganizing, however their approach is specific to the distributed sensor network and would have to be adapted for general purpose worth oriented domains. Other approaches to OSD [3,10,11] tend to completely skim over the problem.

This paper will present and evaluate algorithms for both of the two commonly used approaches to robustness:

1. the *Citizen Approach* [1,12], which involves the use of special monitoring agents (called *Sentinel Agents*) in order to detect agent failure and dynamically startup new agents in lieu of the failed ones.
2. the *Survivalist Approach* [13], which involves using the domain agents to monitor themselves. The domain agents may (a) restart failed agents and (b) create additional replica agents which may take over should the original agents fail.

Our goal is to allow the organization, at its best, to function without any performance degradation in the face of failures. At its worse, the organization should degrade gracefully in proportion to the number of failures.

Note that we do *not* present any new approach to OSD in this paper. Instead we add robustness to the OSD approach presented in [4]. Also, we are *not* trying to develop any new and general approaches to robustness. Instead we were trying to address the robustness issues that arise when using OSD. Our primary contribution is an analysis of the different approaches to robustness when using OSD to design organizations, so that a multiagent user can select the most suitable approach for their application.

The organization of the rest of this paper is as follows. In the next section we discuss the task model that we use for worth-oriented domains. This is followed by a discussion of our approach to OSD. Finally, we evaluate the presented algorithms.

2 Task Model

We use TÆMS as the underlying representation for our tasks (problem instances). TÆMS [14] (Task Analysis, Environment Modeling and Simulation) is a computational framework for representing and reasoning about complex task environments in which tasks (problems) are represented using extended hierarchical task structures [15]. The root node of the task structure represents the high-level goal that the agent is trying to achieve. The sub-nodes of a node represent the subtasks and methods that make up the high-level task. The leaf nodes are at the lowest level of abstraction and represent executable methods – the primitive actions that the agents can perform. The executable methods, themselves, may have multiple outcomes, with different probabilities and different characteristics such as quality, cost and duration. TÆMS also allows various

mechanisms for specifying subtask variations and alternatives, i.e. each node in TÆMS is labeled with a characteristic accumulation function that describes how many or which subgoals or sets of subgoals need to be achieved in order to achieve a particular higher-level goal. TÆMS has been used to model many different problem-solving environments including distributed sensor networks, information gathering, hospital scheduling, EMS, and military planning. [16,17,15,18].

For a more formal description of our task and resource model, please refer to [4,16].

3 Organizational Self Design

In our approach, problem solving requests arrive at the organization continuously at varying rates and with varying deadlines. To gain utility, the agents in the organization need to solve the problems by the given deadlines. The organizational design is directly contingent on the task structure of the problems being solved and the environmental conditions under which the problems need to be solved. Here, the environmental conditions refer to such attributes as the task arrival rate, the task deadlines and the available resources. We assume that all problems have the same underlying task structure, henceforth called *the global task structure*.

To participate in the organization, each agent must maintain some organizational knowledge. This knowledge is also represented using TÆMS task structures, called the local task structures. These local task structures are obtained by rewriting the global task structure and represent the local task view of the agent vis-a-vis its role in the organization and its relationship to other agents. Hence, all reorganization involves rewriting of the global task structure. However, note that the global task structure is *NOT* stored in any one agent, i.e. no single agent has a global view of the complete organization. Instead each agent's organizational knowledge is limited to the tasks that it must perform and the other agents that it must coordinate with — it is this information that is represented using the local task structures.[2]

To allow the agents to store information about other agents in the task structure, we augment the basic TÆMS task representation language by adding organizational nodes. To differentiate organizational nodes from "regular" TÆMS nodes, we refer to non-organizational nodes as *domain* nodes. The four organizational nodes are:

1. **Non-Local-Nodes** are used to represent a domain node in some *other* agent's local task structure. Non-Local-Nodes are used to represent nodes in the global task structure that the agent knows the identity (label) of but does *not* know the characteristics (e.g. quality, cost duration) of[3].
2. **Container-Nodes** are aggregates of domain nodes and other organizational nodes. Container nodes also have a type which determines their purpose. For the purposes

[2] Note that rewriting the local task structure results in a change in the goals and commitments of an agent. After every rewrite of a task structure, the agents that change usually renegotiate the coordination mechanism used to coordinate between themselves. The exact details are beyond the scope of this paper.

[3] At least initially at the time of breakup. It can however learn these characteristics through some coordination mechanism.

of this paper, there are only two types of container nodes: (a) *ROOT* nodes are used to store the high-level goals of an agent and the non-local nodes; and (b) *CLONE* nodes are used to store cloned portions of a TÆMS subtree.

3. **Clone Selectors** are used to select amongst the clones of a node. The purpose of a selector node within a clone-container is to *enable* one or more of the clones, so that the enabled nodes can be "executed" by the agents owning those clones.
4. **NLE-Inheritors** are methods whose sole purpose is to transfer the non-local effect from a non-cloned node to a cloned node or vice versa.

Algorithm 1. CHANGEORGANIZATION

```
1. if ISAGENTOVERLOADED() then
2.     SPAWNAGENT()
3. else if ISAGENTUNDERLOADED() then
4.     COMPOSEAGENT()
5. end if
```

To allow for a change in an agent's organizational knowledge, we define three rewriting operations on a task structure:

Breakup: Breakup involves dividing the local task structure of an agent so that it can be allocated to another agent. When a spawning agent divides a local task structure, A into two subparts B (for itself) and C (for the spawned agent), it still needs to maintain some knowledge about the tasks/methods in C while, at the same time, allowing the spawned agent to have as much autonomy as possible about the execution of C. Specifically the agent will need to know about the subset of nodes in C that are interrelated to the nodes in B, either through NLEs or through subtask relations. These interrelated nodes are represented using Non-Local-Nodes that have the same label as the domain nodes and correspond to the domain nodes. The algorithm for breakup is shown in Algo. 2 and an example is shown in Figure 1. In this example, the task structure represented by Root-1 is broken twice — once at node D to generate Root 2 and a third time at Node E to generate Root 3.

Merging: Merging involves combining two different local task structures from two different agents to form one local task structure. The algorithm for merging is shown in Algo. 3 and an example is shown in Figure 1. The core of the merging algorithm (lines 3 –12) involves finding "overlapping" nodes, i.e. identical nodes that are represented in both the task structures being merged. The merge algorithm goes through all the nodes in the second task structure and tries to find the corresponding nodes in the first task structure. If a corresponding node is found, the algorithm decides which of the two nodes should be kept and which of the nodes should be discarded.

Cloning: Cloning involves creating a complete copy of a substructure, so that it can be allocated to another agent. Cloning serves two purposes: (a) it can be used for load balancing similar to the work of [19]; and (b) it can be used to increase the *robustness capacity* of an agent by having multiple agents work on the same task simultaneously.

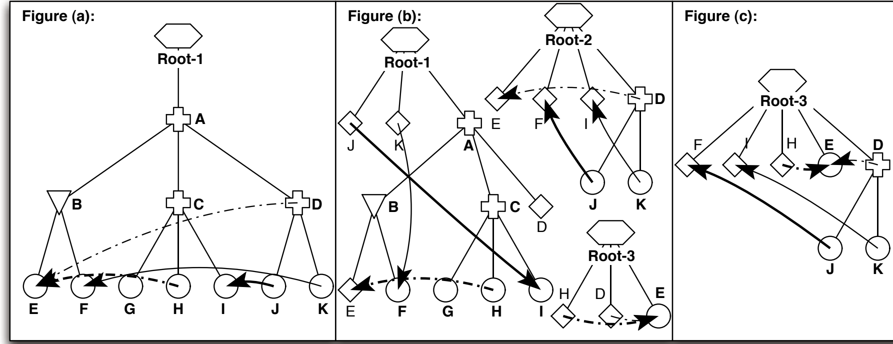

Fig. 1. Task rewriting: *Fig (a)* shows the global TÆMS task structure: The polygons (labelled A – D) represent tasks and the circles (labelled E – K) represent executable methods. The $+$ iconography indicates a *SUM* CAF while $\bigtriangledown$ represents a *MIN* CAF. The arrows represent NLEs — the thick arrows represent hard constraints such as *Enables* (represented by a solid arrow from J to I) and *Disables* (represented by a broken arrow from H to E). The thin arrows show soft constraints such as *Facilitates* (solid arrow from K to F) and *Hinders* (broken arrow from D to E). Method characteristics and other details are omitted. *Fig (b)* shows the breakup of Root-1 at nodes D and E. The diamonds represent non-local nodes, that are the responsibility of some other agent. *Fig (c)* shows the merging of Root-3 and Root 2.

Algorithm 2. BREAKUP (τ, υ)

1. $\overline{\tau} \Leftarrow$ DESCENDENTS(τ) $-$ DESCENDENTS(υ)
2. $\overline{\upsilon} \Leftarrow$ DESCENDENTS(υ)
3. **for all** $\{ N \mid N \in$ NLES$(\tau) \}$ **do**
4.   **if** (SOURCE$(N) \in \overline{\tau}$ **and** SINK$(N) \in \overline{\upsilon}$) **or** (SOURCE$(N) \in \overline{\upsilon}$ **and** SINK$(N) \in \overline{\tau}$) **then**
5. $x \Leftarrow$ GETNONLOCALNODE(SOURCE(N))
6. $y \Leftarrow$ GETNONLOCALNODE(SINK(N))
7. $M \Leftarrow$ COPYNLE(N)
8.   REPLACENODE$(N$, SOURCE$(N), x)$
9.   REPLACENODE$(M$, SINK$(N), y)$
10.   **end if**
11. **end for**
12. $x \Leftarrow$ GETNONLOCALNODE(υ)
13. REPLACENODE(τ, υ, x)
14. **return** CreateRootNode(υ)

The algorithm for cloning is given in Algo. 4 and an example is shown in Figure 2. Lines 4–7 of this algorithm involve creating a copy of all the nodes in the subtree being cloned. Lines 8–21 are used to take care of NLEs in the cloned subtree that have a source or destination as a non-clone node. Such NLEs that transcend clone boundaries have to handled carefully in order to (a) preserve their original semantics and (b) allow the presence of clones to be transparent to the

Algorithm 3. MERGE (τ, υ)

1. **for all** $\{ y \mid y \in$ DESCENDENTS$(\upsilon) \}$ **do**
2. $x \Leftarrow$ FINDNODE(τ, x)
3. **if** NULL(x) **then**
4. DELETENODE(υ, y)
5. ADDNODE(τ, y)
6. **else if** TYPE$(\tau, x) = NonLocal$ **and** TYPE$(\upsilon, y) = NonLocal$ **then**
7. MERGENODES(τ, x, y)
8. **else if** TYPE$(\tau, x) = Local$ **and** TYPE$(\upsilon, y) = NonLocal$ **then**
9. DELETENODE(υ, y)
10. **else if** TYPE$(\tau, x) = NonLocal$ **and** TYPE$(\upsilon, y) = Local$ **then**
11. REPLACENODE(τ, x, y)
12. **end if**
13. **end for**
14. **return** τ

Algorithm 4. CLONE (τ, υ)

1. $\overline{\tau} \Leftarrow$ DESCENDENTS$(\tau) -$ DESCENDENTS(υ)
2. $\overline{\upsilon} \Leftarrow$ DESCENDENTS(υ)
3. $\phi \Leftarrow$ CREATECLONECONTAINER(υ)
4. **for all** $\{ x \mid x \in \overline{\upsilon} \}$ **do**
5. $y \Leftarrow$ COPYNODE(x)
6. ADDNODE(ϕ, y)
7. **end for**
8. **for all** $\{ N \mid N \in$ NLES$(\upsilon) \}$ **do**
9. **if** SOURCE$(N) \in \overline{\tau}$ **then**
10. $x \Leftarrow$ CREATEINHERITINGNODE()
11. ADDNODE(ϕ, x)
12. $L \Leftarrow$ COPYNLE(N)
13. $M \Leftarrow$ COPYNLE(N)
14. REPLACENODE$(N$, SINK(N), $x)$
15. REPLACENODE$(L$, SOURCE(L), $x)$
16. REPLACENODE$(M$, SOURCE(M), $x)$
17. $y \Leftarrow$ FINDNODE$(\phi$, SINK$(M))$
18. REPLACENODE$(M$, SINK(M), $y)$
19. **else if** SINK$(N) \in \overline{\tau}$ **then**
20. {Similar to the source}
21. **end if**
22. **end for**
23. ADDNODE(ϕ, υ)
24. **return** ϕ

non clone nodes. In order to achieve this effect, we create special methods called *NLE-Inheritors*. These methods are simply conduits for the effects from the cloned nodes to the non-clone nodes.

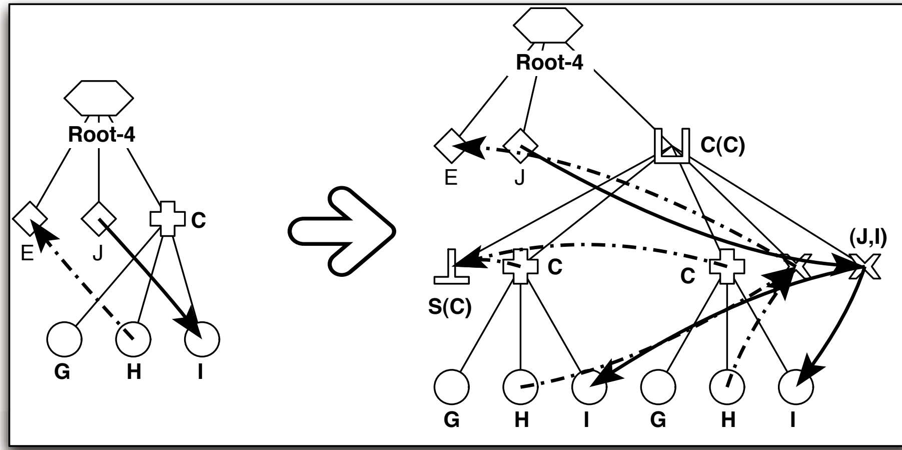

Fig. 2. Task Rewriting (cont.): *Figure showing the cloning of Node C in Root 4*. The $\sqcup$ node (Node C(C)) is used to represent a clone container, a node created for storing the "clones" of a node; the $\perp$ node (Node S(C)) is used to select which clone to "execute" — for the purposes of robustness, all the clones will be executed; finally the $\times$ nodes represent the NLE-inheriting-methods — these nodes form a conduit for NLEs to the non-clone parts of the task structure.

Our approach to OSD involves starting with a single agent responsible for the global task structure (i.e. the local task structure is equal to the global task structure). Each agent in the organization follows the algorithm presented in 1:

– If an agent is overloaded it either breaks or clones its local task view and then spawns a new agent. We define an overloaded agent as one that cannot complete the tasks in its task queue by their given deadlines.
– If the agent is underloaded, on the other hand, it composes with another agent, merging the local task structures of the two organizations. We define an underloaded agent as one that is idle for an extended period of time.

For more details on the precise mechanism used to detect overload and underload please refer to [4].

4 Robustness Mechanisms

Both of our robustness mechanisms involve three parts: (a) monitoring for agent failure; (b) maintaing state information for all the agents; and (c) restarting failed agents.

Furthermore, the underlying mechanism for monitoring and restarting is the same across the robustness mechanisms. Monitoring is achieved by sending out periodic *Are-You-Alive* messages to the set of monitored agents and waiting for *Alive* reply messages. If a reply is not received within a certain interval, we assume that the agent is dead and send a restart message to the environment. The individual mechanisms, however, differ in *who* is responsible for the monitoring and *which* set of agents are monitored.

State information is needed to restart a failed agent. At a minimum, this state information should contain the *organizational state* (i.e. the local task structure) of the agent being restarted. However, the local task information is not sufficient for restarting an agent in a complex domain. The restarted agent will still need information about the *execution context*, i.e. information about the outstanding task instances, information about the methods of a task instance that have already been executed (so that the agent does not try to re-execute them) and information about coordination commitments (because the subtasks have non-local effects and are interdependent on each other). The coordination mechanisms also differ in how they keep a track of this execution context.

4.1 Citizens Approach

The citizens approach involves creating a special monitoring agent (called a *sentinel agent*), which is responsible for all the robustness related responsibilities of the organization. This approach is the simplest to execute — the sentinel agent is the sole monitor that is responsible for monitoring all the agents in the organization. However, to maintain state information, the sentinel needs to listen to *all* the messages exchanged by *all* the other agents, since it needs to store both the set of spawning/composition messages (in order to track the changes to the local task structures of the agents) and the set of execution/coordination messages between the agents (in order to keep a track of the execution context).

Hence, not only does the sentinel effectively becomes a conduit for all the messages, it also has global knowledge about the complete organization — a problem we were trying to avoid by using the OSD approach in the first place. Furthermore, the sentinel can (a) quickly become overwhelmed by all the messages that it needs to track and (b) become a central point of failure[4]. The solution might be to add multiple sentinel agents — we will now need to create an organization for the sentinels (for which we could, again, use OSD) and a way of monitoring the monitors.

Hence, we focus on developing algorithms for the survivalist approach and use the citizens approach for comparison.

4.2 Survivalist Approach

In the survivalist approach, there are no special agents responsible for monitoring and restarting failed agents. Instead the domain agents divide the monitoring responsibilities amongst themselves. Furthermore, some/all domain agents may be replicated in order to (a) increase the robustness capacity of the organization; (b) decrease the response time to a failure, and (c) process task instances in parallel, thus helping to balance the load.

The obvious advantage of the survivalist approach is that no one agent is overburdened with the monitoring responsibilities. Also there is no central point of failure and no agent with global knowledge of the organization. Furthermore, the survivalist approach can take into account the interplay between a satisficing organizational structure and probability of failure. For example, one way of achieving a higher level of robustness in the survivalist approach, given a large numbers of agent failures, would be to

[4] It's unreasonable to assume that the other agents might fail, but the sentinel will never fail.

relax the task deadlines. However, such a relaxation would result in the system using fewer agents in order to conserve resources, which in turn would have a detrimental effect on the robustness. These advantages come at a cost of increased complexity of the monitoring mechanism.

Creating a monitoring set of agents. The monitoring set of an agent, *Agent A*, is defined as the set of agents that are responsible for monitoring *Agent A* for failures. We assume that the minimum cardinality of this set, *N* is an input to the organization[5]. Also in our approach, all monitoring is mutual, i.e. if *Agent A* is in the monitoring set of *Agent B* (i.e. if *Agent A* is responsible for monitoring the health of *Agent B*), then *Agent B* is in the monitoring set of *Agent A*. This is by design, because *Agent A* on receiving an *are-you-alive* request from *Agent B*, already knows that *Agent B* is alive and does not need to send *Agent B* a separate request.

Each agent is responsible for determining its monitoring set. At the time an agent, say *Agent A*, is first spawned, it runs the following algorithm:

1. *Agent A* determines its related set. The related set of *Agent A* is the set of agents that have a coordination relationship with *Agent A*. (This coordination relationship would exists because of interdependent tasks and NLEs in the task structures of the agents).
2. If the number of agents in the related set of *Agent A* is greater than N, *Agent A* sends a message to each of the related agents, requesting the cardinality of their respective monitoring sets, and then goes to Step 3. If this number is less than N, *Agent A* adds all of the related agents to its monitoring set and then jumps to Step 4.
3. *Agent A* picks the N related agents with the lowest monitoring-set cardinalities to be in its monitoring set.
4. *Agent A* sends messages to all the other non-related agents, requesting their monitoring-set cardinalities. (This can be done using a single broadcast message). *Agent A* then iteratively selects agents with the lowest monitoring-set cardinalities until either (a) it has N agents in its monitoring set or (b) until all the agents have been exhausted (i.e. there are less than N agents in the whole organization).

Finally, once *Agent A* has determined its monitoring set, it can send a message to each of the agents in its set requesting them to monitor its health. In addition to being distributed, other advantages of this algorithm are: (a) Steps 1–3, can be piggy-backed onto the coordination-mechanism negotiation messages exchanged with the agents in the related set and (b) This scheme will reduce the frequency of *are-you-alive* messages transmitted since the agents will be communicating in-band as a part of their normal tasks processing.

Augmenting the robustness capacity of an organization. The robustness capacity of an organization is defined as the number of agent failures that an organization can withstand. The robustness capacity is equal to the kill count minus 1, where the kill count is the *minimum* number of agent that need to be killed in order to kill the organization

[5] It should be possible to develop an algorithm for learning the optimal value of N given the environment conditions — i.e. the probability of failure. We plan to incorporate this into our future work.

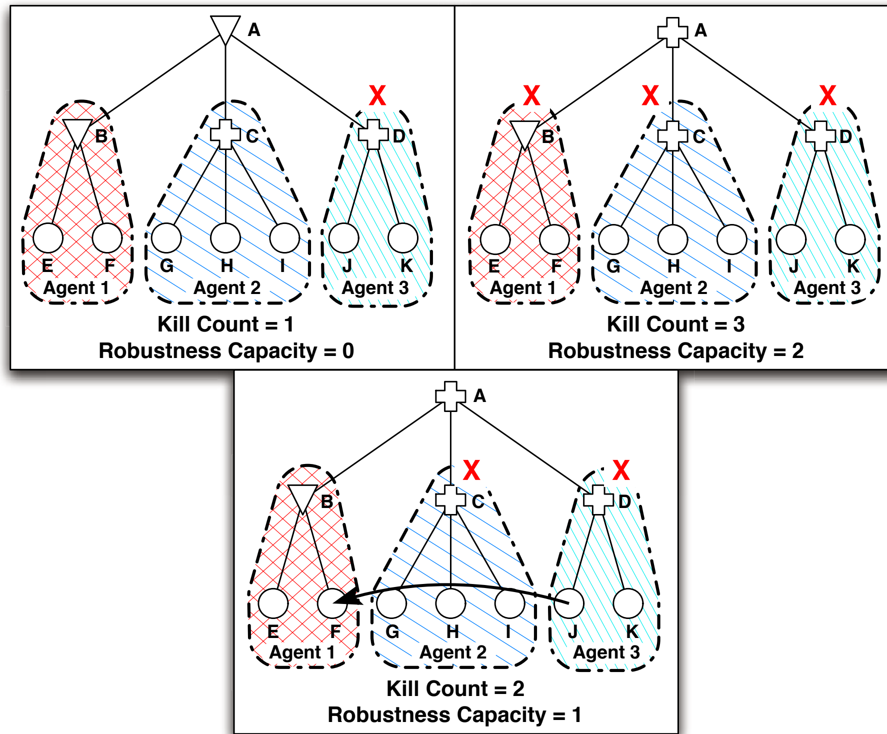

Fig. 3. Computing Robustness Capacity: Figure showing how the global task structure and its breakup amongst the agents affects the robustness capacity. The crosses on the agents show which agents need to be killed in order to kill the organization.

(i.e. ensure that the remaining agents cannot complete tasks without having to restart more agents).

The robustness capacity of an organization is dependent on (a) the underlying global task structure and (b) the way it has been divided amongst the agents. The CAFs of the global task structure, especially the root CAF, determines the number of alternatives available for achieving a task. For example, a one-level deep task structure with a *MAX* CAF and three subtasks would have three alternative ways of achieving the task. If each of these three alternatives was divided amongst three agents, the resultant organization would have a kill count of 3 and a robustness capacity of 2.

Figure 3 shows how the global task structure and its breakup amongst the agents affects the robustness capacity of an organization. The first task structure has a *MIN* CAF as its root, so *either* of Agents 1, 2 or 3 can be killed in order to kill the organization. The kill count is 1 and, hence, the robustness capacity is 0. The second task structure has a *SUM* as its root, so *all* the three agents need to be killed in order to kill the organization. Hence, the kill count is 3 and the robustness capacity is two. The third task structure is similar to second one except that it has an enablement from Method J to F. With the task structure divided amongst the agents as shown, if Agent 3 is killed, there is no way

to complete Method J. This, in turn, means that Method F will never be enabled, i.e. the quality of Method F will always be 0. Since Task B has a *MIN* CAF, the quality of Task B will also be 0 and, as a result, Agent 1 has effectively been poisoned. Hence, only Agents 2 and 3 need to be killed to kill the organization and the kill count is 2.

Augmenting the robustness capacity of an organization is the process of adding agents to the organization so as to increase its kill count. Again, we are assuming that the desired kill count, *K* is an input to the organization. A trivial way to do this would be to replicate each agent *K-1* times. However, this would be inefficient as it does not take into account the existing kill-count of the organization.

The first step towards increasing the robustness capacity of an organization would be to compute the existing kill-count, and then to "add" agents by breaking up the global task structure and spawning agents in a way that increments this kill-count. Unfortunately, the bad news is that computing the kill-count of an organization based on an underlying TÆMS task structure is NP-hard. An informal proof follows:

This proof is based on the reduction of a minimum set covering problem to a TÆMS based organization, where the kill-count of that organization would be the solution to this problem. Assume a ground set *M* consisting of m elements, $\{e_1, e_2, ..., e_m\}$ and n subsets $\{s_1, s_2, ..., s_n\}$. Create a TÆMS task structure, with a *MAX* CAF as the root and the subsets, $\{s_1, s_2, ..., s_n\}$ as its subtask nodes. Finally replace each node s_i with a *MIN* CAF task, the subtasks of which will be the methods, $\{e_{i,1}, e_{i,2}, ..., e_{i,j}\}$, where each method corresponds to an element of s_i. Finally, assign m agents to the organization, where each method corresponding to e_i is assigned to agent a_i. This reduction will provably take polynomial time.

Since, (a) the problem of computing the kill count/robustness capacity of a problem is NP-hard and (b) augmenting the organization by spawning agents at specific places will interfere with other desirable characteristics such as balancing the execution time and maximizing quality, we chose an alternative approach to augmenting the robustness capacity.

In our approach, the initial root node of the global task view is cloned $K-1$ times and each clone is allocated to a separate agent. These agents are responsible for individually forming their own independent organizations and spawning and composing with agents independently.

Frequency of *are-you-alive* messages. Ideally, we want each agent in the monitoring set of an agent A to send an *are-you-alive* request at a different time. To achieve this, we initialize each agent with a random seed. The *next-poll-time* is initialized to the *poll-interval* plus this random seed. Also the next-poll-time is recalculated on receiving *any* message from the monitored agent.

5 Evaluation

To evaluate the two robustness mechanisms, we ran a series of experiments that simulated the operation of the OSD organization when those mechanisms were employed. We tested the performance of the survivalist approach against the citizens approach with the following (per agent/per cycle) probabilities of agent failures: 0.000, 0.002, 0.006

and 0.010. Here a probability of 0.006 means that on every clock cycle, each agent has a 0.6 % chance of failing. Note that despite these seemingly low probabilities of failure, the rate of failure is actually greater than can be expected for any real world application. For example, a probability of failure of 0.006 implies that *every* agent can be expected to fail 15 times during a 2500 cycle run.

We used a randomly generated TÆMS task structure with a maximum depth of 4, branching factor of 3, and NLE count of 10 to seed the experiments. We were careful to use the same task structure, task arrival times, task deadlines and random numbers for each of the (robustness-mechanism, failure probability) pairs. Each experiment was repeated 15 times using a different randomly generated task structure. The experiments were run for 2500 clock cycles. For the survivalist approach we used 3 as the value of K, i.e. the root node is cloned *twice*. We used the following performance criteria to evaluate our two approaches to robustness:

1. The average number of agents used.
2. The number of tasks completed.
3. The average turnaround time. The turnaround time is defined as the difference between the time at which a task is either completed or failed and the time at which the task was generated (the generation time). The average turnaround time is the turnaround time divided by the total number of tasks.
4. The average quality accrued. The average quality is defined as the total quality accrued during the experimental run divided by the sum of the number of tasks completed and the number of tasks failed.
5. The total messages sent by all the agents.
6. The total resource cost of the organization.

The average results for these measured performance criteria are shown in Figure 4.

We also tested the statistical significance of the obtained results using the *Wilcoxon Matched-Pair Signed-Rank* tests with $p < 0.05$. *Matched-Pair* signifies that we are comparing the performance of each robustness approach on precisely the same randomized task set, environmental conditions and failure probabilities within each separate experiment. Some interesting observations are:

- We were pleasantly surprised by the overall performance of these approaches. In particular there was no statistical difference between the number of tasks completed in the absence of agent failure and all the situations in which the citizens approach was employed. Only the survivalist approaches at probabilities of failure of 0.006 and 0.010 performed statistically worse than the no failure case. This result shows that the survivalist approach is a credible distributed alternative to the citizens approach.
- As expected the fewest number of agents were used when the probability of failure was 0. Surprisingly, not only did the survivalist approach use fewer agents than the citizens approach, this result was *statistically significant* for failure probabilities of 0.006 and 0.010. This results is surprising because we expected the survivalist approach to use more agents since the survivalist approach pro-actively replicates agents to increase the robustness capacity of the organization. The reasoning behind this result becomes obvious once we look at the percentage of the total tasks completed by the survivalist organizations. The survivalist organizations complete

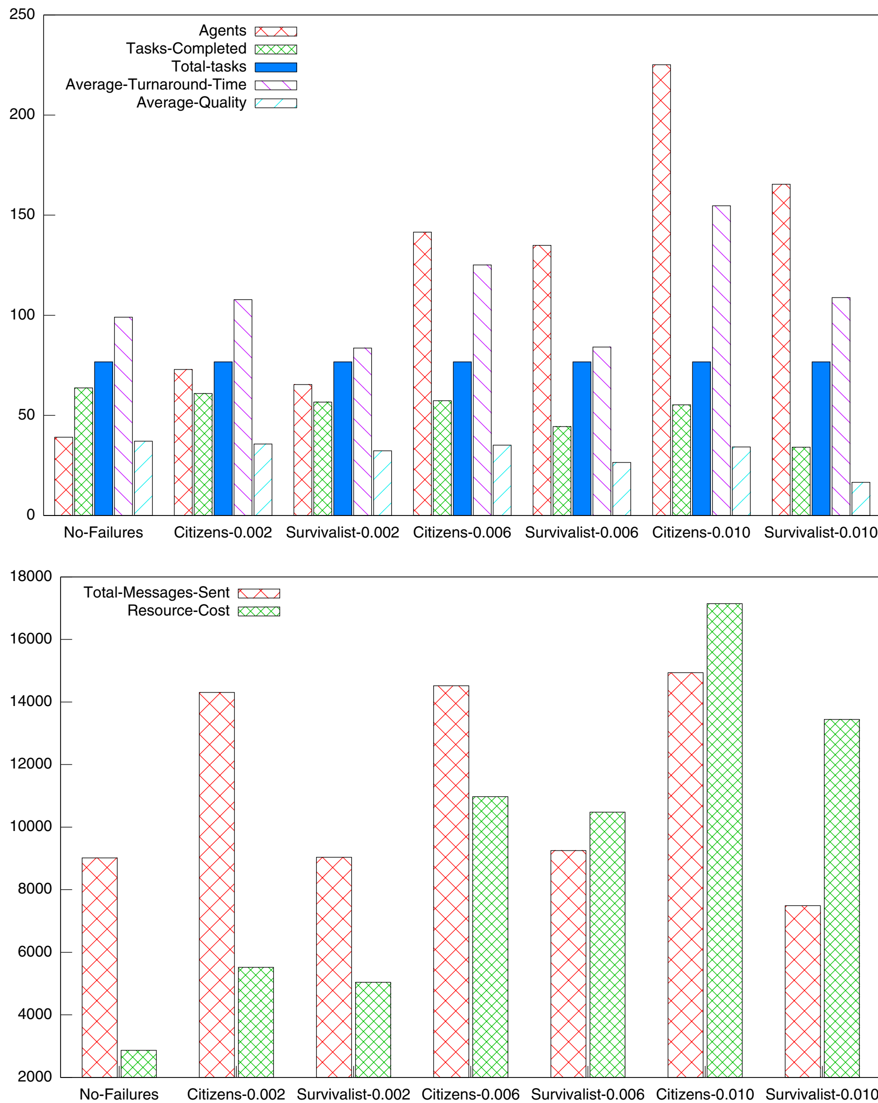

Fig. 4. Graph showing the various measured parameters for the different robustness mechanisms. The numbers below the mechanisms indicate the probability of agent failure per agent per cycle.

fewer tasks for failure probabilities of 0.006 and 0.010. This is because the survivalist organizations are being overwhelmed by the failure rate – the agents are failing much faster than the rate at which the organization can detect the failures and restart the failed agents.

Using a higher value of K, should allow the survivalist approach to perform as well as the citizens approach. As a part of our future work, we would like to automatically determine the best value of K at run-time.

- The citizens approach used a significantly larger number of messages to achieve similar levels of robustness. This is primarily because of the monitor agent has to record all the messages exchanged by the agents.
- Finally, the turnaround time is statistically significantly lower for the survivalist approaches with probabilities of 0.002 and 0.006 than all the other approaches including the no failure approach. This is probably the result of extra agents better balancing the load of the organization.

6 Conclusion

This paper was primarily concerned with the robustness of organizations, generated at run-time through the use of organizational-self design (OSD), in the presence of agent failures. We have incorporated the two commonly used approaches to robustness, that is, the citizens approach and the survivalist approach into our OSD system. The citizens approach is simpler and more effective than the survivalist approach but suffers due to the use of a single centralized and omniscient monitoring agent to achieve its robustness. The survivalist approach, on the other hand, is truly distributed and we have shown it to be a credible alternative to the citizens approach since it uses fewer agents and fewer (communication) resources to achieve similar levels of robustness.

In our future work, we would like to develop a more fine-grained method for augmenting the robustness capacity of an organization in the survivalist approach — one that will use some heuristic to compute or *underestimate* the current robustness capacity of the organization and then augment the robustness capacity by cloning specific agents. We would like to see if such a fine-grained approach will use even fewer agents than the currently presented survivalist approach.

References

1. Dellarocas, C., Klein, M.: An experimental evaluation of domain-independent fault handling services in open multi-agent systems. In: Proceedings of the International Conference on Multi-Agent Systems (ICMAS 2000) (July 2000)
2. Smith, R.G.: The contract net protocol: High-level communication and control in a distributed problem solver. In: Distributed Artificial Intelligence, pp. 357–366. Morgan Kaufmann Publishers Inc., San Francisco (1988)
3. Ishida, T., Gasser, L., Yokoo, M.: Organization self-design of distributed production systems. IEEE Transactions on Knowledge and Data Engineering 4(2), 123–134 (1992)
4. Kamboj, S., Decker, K.S.: Organizational self-design in semi-dynamic environments. In: AAMAS 2007, pp. 1220–1227 (2007)
5. Anderson, D.P.: Boinc: A system for public-resource computing and storage. In: Fifth IEEE/ACM International Workshop on Grid Computing (GRID 2004), pp. 4–10 (2004)
6. Shirts, M., Pande, V.S.: COMPUTING: Screen Savers of the World Unite! Science 290(5498), 1903–1904 (2000)
7. Wagner, T., Raja, A., Lesser, V.: Modeling uncertainty and its implications to sophisticated control in taems agents. Autonomous Agents and Multi-Agent Systems 13(3), 235–292 (2006)
8. DeLoach, S., Oyenan, W., Matson, E.: A capabilities-based model for adaptive organizations. Autonomous Agents and Multi-Agent Systems 16(1), 13–56 (2008)

9. Sims, M., Goldman, C.V., Lesser, V.: Self-organization through bottom-up coalition formation. In: AAMAS 2003, pp. 867–874. ACM Press, New York (2003)
10. So, Y., Durfee, E.H.: Designing tree-structured organizations for computational agents. Computational and Mathematical Organization Theory 2(3), 219–245 (1996)
11. Frederic, G., Jacqueline, A.: Logical reorganization of DAI systems. LNCS. Springer, Heidelberg (1995)
12. Klein, M., Rodriguez-Aguilar, J.A., Dellarocas, C.: Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. Journal for Autonomous Agents and Multi-Agent Systems 7(1-2), 179–189 (2003)
13. Marin, O., Sens, P., Briot, J., Guessoum, Z.: Towards adaptive fault tolerance for distributed multi-agent systems. In: Proceedings of European Research Seminar on Advances in Distributed Systems (ERSADS 2001) (May 2001)
14. Lesser, V.R., Decker, K., Wagner, T., et al.: Evolution of the GPGP/TÆMS Domain-Independent Coordination Framework. Autonomous Agents and Multi-Agent Systems 9(1-2), 87–143 (2004)
15. Chen, W., Decker, K.S.: The analysis of coordination in an information system application - emergency medical services. In: Bresciani, P., Giorgini, P., Henderson-Sellers, B., Low, G., Winikoff, M. (eds.) AOIS 2004. LNCS, vol. 3508, pp. 36–51. Springer, Heidelberg (2005)
16. Decker, K.S.: Environment centered analysis and design of coordination mechanisms. Ph.D. Thesis, Department of Computer Science, University of Massachusetts, Amherst (May 1995)
17. Decker, K.S., Li, J.: Coordinating Mutually Exclusive resources using GPGP. Autonomous Agents and Multi-Agent Systems 3(2), 133–157 (2000)
18. Zimmerman, T.L., Smith, S., Gallagher, A.T., Barbulescu, L., Rubinstein, Z.: Distributed management of flexible times schedules. In: Sixth AAMAS (May 2007)
19. Shehory, O., Sycara, K., Chalasani, P., Jha, S.: Agent cloning: an approach to agent mobility and resource allocation. IEEE Communications Magazine 36(7), 58–67 (1998)