Implementing Fault-Tolerant Services in Goal-Oriented Multi-Agent Systems

Sebnem BORA
Department of Computer Engineering, Ege University, Izmir, Turkey
sebnem.bora @ege.edu.tr

***Abstract*—In this paper, findings and analysis detail the implementation of fault tolerance services into a goal-oriented multi-agent systems development platform. Fault tolerance services are used to provide replication-based fault tolerance policies (i.e. static and adaptive) to multi-agent systems. This approach provided flexibility and reusability to multi-agent systems because fault tolerance policies were implemented as reusable plan structures. Thus, whenever an agent was needed to be made fault-tolerant, plans for fault tolerance policies were simply activated by sending a request message.**

***Index Terms*—availability, fault tolerance, multi-agent systems, replication, redundancy, software agents.**

I. Introduction

Multi-agent systems (MAS) are vulnerable to failures that can occur when a system crashes and/or there are shortages of system resources. In addition, MAS is susceptible to failures when communications links are either interrupted or disconnected. Moreover, MAS is directly affected by errors in programs. Therefore, any fault in an agent can be disseminated thus creating systems failures in the MAS. It would appear that fault tolerance is a necessary paradigm that must be taken into consideration for the multi-agent development environment.

There are many different types of multi-agent systems development platforms; however, only a limited number of platforms offer fault tolerance [1-9]. Those multi-agent systems development platforms provide beneficial solutions to the problem of fault tolerance in multi-agent systems. However, certain techniques used for troubleshooting/solving specific problems can force a specific multi-agent system to become less flexible and unreusable.

The basis of this research is to provide fault tolerance to MAS by implementing replication-based fault tolerance policies as reusable agent plans. The details of static and adaptive fault tolerance policies are embedded into reusable agent plans using Hierarchical Task Network (HTN) formalism [10]. By doing so, fault tolerance policies can be implemented as distinct plans. Hence, an agent is able to change its fault tolerance policy autonomously by changing its fault tolerance plans. Moreover, whenever an agent needs to be made fault-tolerant, plans are activated by sending a request message. That message is sent to the agent by a user or another agent to make it fault-tolerant. Even if these plans are not included in the plan library, they would still be sent to the agent to be fault-tolerant.

In order to apply a static fault tolerance policy in MAS, the replication degree and strategy are defined before the application starts. In adaptive fault tolerance policies, the replication degree and strategy are defined at runtime. Data illustrate the most effective approach is to apply the static fault tolerance policy to less critical agents, agents that exhibit predictable behaviors, and whose organizational structure does not change at run time. In addition, it is necessary to apply adaptive fault tolerance policies to the critical agents or agents having their criticalities understood during the organization's lifetime, and the organization that its structure changes during run time. Both fault tolerance policies are simultaneously applied to different replica groups of a multi-agent system [11].

The necessary services used by fault tolerance policies are identified and how these services are implemented and then integrated into a multi-agent system development framework are explained. The services to support replication-based fault-tolerant systems such as replication service, group communication service, and membership service are essential in constructing the infrastructure for supporting fault tolerance policies.

The remainder of this paper is structured as follows: Section 2 presents the context of the research and introduces an abstract architecture for applying fault tolerance in MAS and the SEAGENT platform [12-13] (A Semantic Web Enabled Multi-Agent Development Framework) architecture in which fault tolerance services are developed; Section 3 describes how to implement fault-tolerant services for a multi-agent system using the SEAGENT platform; Section 4 presents a performance analysis; and lastly Section 5 provides the conclusion.

II. Context of This Work

In any computer system, both processes and communication channels sometimes do fail. As a result, they depart from desirable behaviors. The failure model defines ways in which failure may occur in order to provide a more accurate understanding of the effects of the failures. In this work, the failure model is defined as fail-silent model where the considered system allows only crash failures [14].

A. Types of Fault Tolerance Techniques Considered

A number of design techniques have been proposed in the literature to implement fault-tolerant systems. Specifically, replication-based techniques that mask process or agent failures from the users of the system are considered here. There are two main techniques for replication: the passive replication technique and the active replication technique.

Digital Object Identifier 10.4316/AECE.2014.03015

In active replication, there are multiple copies of a service or an agent (called replicas) processing requests sent by clients and synchronizing internal states with all other replicas. Group communication provides multi-point-to-multi-point communication by organizing replicas in groups [15]. If a primary service or agent fails, any replica can become a primary service provider.

In the passive replication technique, there are multiple copies of a service or an agent (replicas); however, primary services (leader) only respond to a client's requests. Also, leaders periodically update the replicas' states. If a leader fails, a replica can be elected as a new leader.

The replication service of a fault-tolerant multi-agent system can execute an active or passive replication technique. Active and passive replication techniques mainly focus on coordination within a group. In addition to coordination requirements, the replication degree, or the number of replicas within a group, is a crucial concept for applying replication-based fault tolerance policies. One key challenge is to understand how the system will decide on the number of replicas at runtime. The replication degree can be identified adaptively or statically. In a static fault tolerance policy, the replication degree is set by a programmer during the initialization period. In an adaptive fault tolerance policy, the leader agent determines the replication degree based on the system's resources. This process employs an observation service that observes the agents' behaviors as well as the availability of resources, and adaptively reconfigures the system's resources [11].

There are many different systems and research projects that have proposed a range of services to support replication-based fault-tolerant systems [16-18] such as global time service, replication service, group communication service, failure detector and membership service. These services form the infrastructure in order to enhance fault tolerance in distributed and multi-agent systems.

B. An Agent-Based Architecture

In the following section, a rough sketch of the proposed architecture is presented for providing fault tolerance to MAS. This architecture is built on a goal-oriented MAS architecture. The MAS architecture is supplemented with the Foundations for Physical Agents' (FIPA) software standards specifications for agent based systems.

The main modules of a goal-oriented agent [19] are its goal manager and planner. A goal manager identifies the goal extracted from an incoming FIPA-ACL message and initializes the plan (a partially-ordered sequence of primitive tasks) of this goal. Next, a planner schedules and executes the plan in a planning formalism [20]. If a users' aim is to integrate services for supporting replication-based fault-tolerant systems into a goal-oriented FIPA-based architecture, then he/she has to identify new goals coming from fault tolerance requirements, plans of these goals, and reusable services that can be used by plans of these goals.

A replication-based fault tolerant MAS consists of leader agents and their replicas that run on computers. In any event that a computer crashed or is disconnected from a network, leader agents and their replicas could fail.

In Fig. 1, a leader agent and its replica are presented. The aim of the leader agent is to be fault-tolerant against any crash failures; therefore, it has the *Fault Tolerance* plan and *Adaptive Fault Tolerance* plan in which adaptive replication is performed. In addition, it uses necessary services such as the failure detector, the membership service, the group communication service, the replication service, and its subservices such as the cloning service and the leader election service.

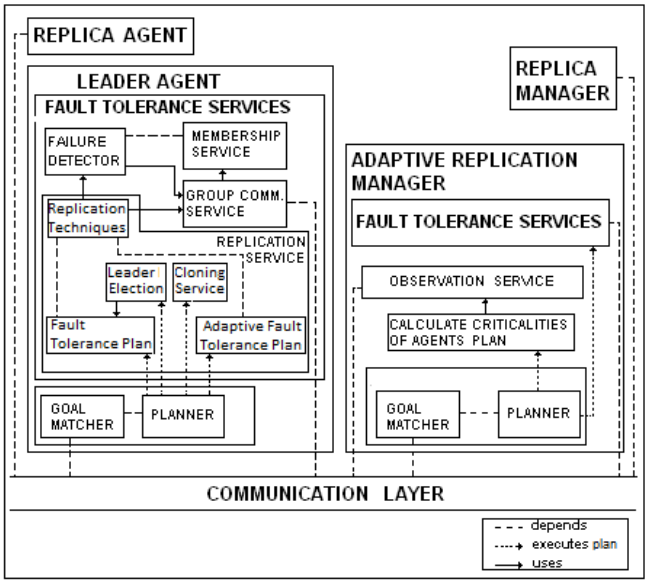

Figure 1. An abstract architecture of a replication-based fault tolerant MAS

In a static fault tolerance policy, the main goal is to keep the group's replication degree constant unless a request is received for altering it. If a failure report (i.e., it reports a replica has crashed) is received from the failure detector, the *Fault Tolerance* plan is executed by the leader in order to reach the same replication degree when the application starts. The *Fault Tolerance* plan uses the cloning service for replication of new replicas in order to keep the replication degree constant.

In order to apply an adaptive fault tolerance policy, the environment must be monitored to collect data and then data must be analyzed to adapt agent systems. Therefore, an agent called the adaptive replication manager (illustrated in Fig. 1) is used. The goal of the adaptive replication manager is to dynamically compute the criticalities of agents applying the adaptive fault tolerance policy in an agent system and initiate the *Calculate Criticalities of Agents* plan to achieve this goal. The quantities that define the criticalities of agents are used for calculations to share the limited resources between agent replica groups. In the *Calculate Criticalities of Agents* plan, an observation service is used.

The observation service collects data from the environment and then processes it to identify critical agents. The quantity that defines a level of agent's criticality is sent within the content of a FIPA-ACL message to the leader agent(s) applying the adaptive fault tolerance policy. When the leader receives this message, it executes the *Adaptive Fault Tolerance* plan. Thus, the leader creates new replicas or removes replicas with respect to the content of the FIPA-ACL message received from the adaptive replication manager.

In order to provide fault tolerance to multi-agent systems, the group communication service, the membership service,

the failure detector, the observation service, and the replication service were implemented in the SEAGENT Platform. In Section 3, it is explained in detail how these services are implemented. In the next section, SEAGENT layered architecture is presented.

C. Layered Architecture of the SEAGENT Platform

In order to integrate fault tolerance services into the SEAGENT platform, SEAGENT's layered software architecture needs to be introduced briefly.

The initial layer of the platform architecture is the platform's communication infrastructure which incorporates FIPA's Agent Communication and Agent Message Transport specifications for agent messaging. This layer transmits messages by using FIPA-ACL; however, it only supports FIPA RDF content language in order to carry semantic web enabled content [12-13]. FIPA standards are not concerned with the message transport protocol inside an agent platform. Agents in the SEAGENT platform communicate by using Java Remote Method Invocation (RMI). Moreover, Internet Inter ORB Protocol (IIOP) is used to communicate with different agent platforms. The realization of this protocol uses Java RMI over IIOP.

The second layer of SEAGENT architecture includes packages in order to construct the functionality of the platform. For example, the Agency package manages the internal functionality of an agent like an *agent operating system*. It includes dispatcher, matcher, scheduler and executer modules. Each module runs concurrently as a separate Java thread and uses common data structures. The dispatcher module sends outgoing messages and retrieves incoming messages. It resolves the incoming FIPA-ACL message into a new objective, puts it in the objective queue and notifies the matcher. Next, the matcher module matches the incoming objective to a plan. Then, the scheduler module determines each task's execution time. The scheduler and executer modules use the waiting and ready queues as common data structures. If a task is ready to be executed, the scheduler deletes it from the waiting queue and places it in the ready queue. The executor module then executes the ready tasks [12-13].

In the Core Functionality Layer, there are service sub-packages which form standard MAS services such as the Semantic Service Matcher, the Directory Facilitator (DF) Service, the Ontology Management Service, and the Agent Management Service (AMS) [12-13].

The third layer of the overall architecture includes generic agent plans that are divided into two generic packages: Generic Behaviors and Generic Semantic Behaviors. The Generic Behavior package contains independent domain reusable behaviors such as *Calculation of Criticalities of Agents, Leader Election, Fault Tolerance, Adaptive Fault Tolerance, and Cloning Service* (*Replication Service)* behaviors for fault-tolerance. The Generic Semantic Behaviors package only contains the semantic web related behaviors. In the following section, it is explained in detail how to implement fault tolerance services.

III. IMPLEMENTING FAULT TOLERANCE SERVICES IN THE SEAGENT PLATFORM

In order to implement fault-tolerant services in the SEAGENT Platform, specific services carried out in the Agency Package that could cause performance bottleneck were identified if implemented as reusable plans.

For example, when a failure detector (illustrated in Fig. 1) was implemented as a reusable plan, the scheduled actions were not executed in a timely manner due to the multi-threaded structure in the SEAGENT platform. The actions were held in the waiting and ready queues of the scheduler and executer modules delayed by the execution of other tasks; therefore, when new messages arrived, the dispatcher's thread started to execute its task. As a result, the actions of the periodic tasks could not be executed on time. For example, the heartbeat mechanism of the failure detector had to periodically multicast alive messages. However, when the time came for the heartbeat mechanism of the failure detector to send alive messages, some requests were fetched by the dispatcher. Consequently, the heartbeat mechanism was not able to send alive messages on time because of the requests needed to be processed. It is evident that implementing the failure detector and the services that required periodic tasks created a performance bottleneck in the case of modeling them as reusable plans.

A membership service maintains a list of agents which are currently in a replica group. It depends on a failure detector to reach a decision about the group's membership. The membership service and the failure detector (illustrated in Fig. 1) were implemented in the SEAGENT's Agency Package. Thus some functions and data structures are shared without introducing extract communication overhead.

SEAGENT also supports a multicasting feature that can implement the group communication service (highlighted in Fig. 1). However, this basic multicasting mechanism delivers requests in an arbitrary order. In multicasting, an ordering sequence must be provided to fault-tolerant systems since consistency between replicas is built by performing incoming requests in order. The total ordering scheme provides the ordering sequence and is implemented in the Agency package to provide an ordered multicasting mechanism of the group communication service.

As aforementioned in previous sections, the adaptive replication manager's responsibility is to compute the agent's criticality and associates with the replication service to dynamically adapt resource allocation. It has the *Calculate Criticalities of Agents* plan (illustrated in Fig. 1) which is implemented in the Generic Behaviors Package of the Reusable Behavior Layer of SEAGENT. The *Calculate Criticalities of Agents* plan uses an observation service that is implemented as a reusable plan in the Generic Behaviors Package.

The main priorities of the replication service are to create new replicas, destroy agents and apply fault tolerance policies such as static and adaptive ones. The replication service provides certain subservices such as cloning [21-22], and uses other services such as membership service, group communication service, and failure detector (illustrated in Fig. 1) in order to achieve its purposes. The replication services' internal mechanisms change in accordance with the replication techniques, such as active and passive replication techniques. The replication service is implemented as reusable plans in the Generic Behaviors Package of SEAGENT. Thus, this makes the agents flexible

in terms of fault tolerance, since they are able to change a preset plan at run-time via a FIPA-ACL message.

When fault tolerance services are integrated into the SEAGENT platform, an agent can be replicated with different strategies. Each replica group has a leader which coordinates the group and communicates with other agents. When a leader fails, a new replica is selected as a new leader in the replica group.

The next section presents how the membership service and failure detector are implemented in the SEAGENT's agency package.

A. Implementing Failure Detector and Membership Service in the Agency Package

The membership service employs a failure detector to determine the group's membership. The decision made by the membership service includes a *view* which is a list of replicas of a group. Reliable multicast services multicast messages to members in a current view.

In this study, the failure detector was implemented by using an unreliable failure detector approach [23]. Each agent has a failure detector that exchanges heartbeats periodically; therefore, the dispatcher module uses timers to implement the failure detection mechanism. One of the timers (Heartbeat_Timer) periodically multicasts a heartbeat message to group members. Definitions of methods and data structures used for implementing the failure detector and a pseudo code of Heartbeat_Timer are presented in Fig. 2 and Fig. 3, respectively.

Definitions:

heartbeat_no: It is the number of alive message sent by an agent to the group.
SEND_MESSAGE(Performative, act, argument, receiver): A message is prepared as a FIPA-ACL message which contains its **Performative**, its **receiver**, and its content including its **act**, and its **argument**.
Heartbeat_Timer(Agent, Period): Agent sends "Alive" in every **Period**.
RECEIVED_MESSAGE(act, argument, sender): A FIPA-ACL "INFORM" message is received that it contains its **sender** and its content including its **act**, and its **argument**.
heartbeat_data_structure(Agent): It stores information related to the **Agent**'s timeout and heartbeat_no.
get_heartbeat _no (Agent): It returns the Agent's heartbeat_no.
get_timeout(Agent): It returns the Agent's timeout .
membership_data_structure(Agent): It stores the **Agent**'s timeout, suspect_vector and delete_member_vector.
suspect_vector(Agent): It stores the names of agents that suspect **Agent**.
delete_member_vector (Agent): It stores the names of agents that send the"delete" **Agent** messages.
Failure_Detector_Timer(Agent, time, number): It compares a **number** to received heartbeat_no in order to determine whether it has received any new alive message(s) from **Agent** during a period of **time**
DELETE_MEMBER(Agent): Agent is deleted from membership_data_structure and heartbeat_data_structure.
Failure_state(Agent): Defines the state of the **Agent** as "SUSPECT" or "UNSUSPECT".

Figure 2. Definitions used for implementing the failure detector

Periodic Heartbeat_Timer(AgentA, Period)
▷ **AgentA** sends alive in defined **Period**.
1 At initial: heartbeat_no ← 0
2 SEND_MESSAGE ("INFORM","ALIVE", heartbeat_no, group)
3 heartbeat_no ← heartbeat_no +1

Figure 3. Algorithm for Heartbeat_Timer of the failure detector

A heartbeat message is prepared as a FIPA-ACL message and contains a performative (INFORM), which includes the name of the sender, the names of the receivers, the type of language, the contents of the act, the actor, and the number of heartbeats as an argument. The message includes FIPA RDF content language. A heartbeat message sent by AgentA is as follows:

```
(
INFORM
        :Sender  AgentA
        :Receiver          all members of the group
        :Language          FPA RDF 0
        :Content
<rdf:RDF
...
   <j.0:act>alive</j.0:act>
   <j.0:actor>EtmenA@aegeants.com</j.0:actor>
   <j.0:argument>0</j.0:argument>
   </rdf:Description>
...
...
</rdf:RDF>
)
```

When a heartbeat message is received by agents in a group, each agent's timer (called Failure_Detector_Timer) begins (line 4 in the failure detector's algorithm illustrated in Fig. 4). Next, each Failure_Detector_Timer waits to receive a new heartbeat message sent from the same agent that sent the previous heartbeat message.

FAILURE DETECTOR
▷AgentA evaluates incoming messages related to fault tolerance.
1 if(RECEIVED_MESSAGE ("ALIVE", heartbeat_no, AgentB))
2 heartbeat_data_structure(AgentB)← heartbeat_no
3 timeout←get_timeout(AgentB)
4 Failure_Detector_Timer (AgentB, timeout, heartbeat_no)
5 if(RECEIVED_MESSAGE ("SUSPECT", AgentB, AgentC))
6 membership_data_structure ← suspect_vector(AgentB) ← AgentC
7 if(all members suspect AgentB)
8 SEND_MESSAGE("INFORM","DELETE", AgentB, group)
9 if(RECEIVED_MESSAGE ("DELETE", AgentB, AgentC))
10 membership_data_structure ← delete_member_vector (AgentB) ←AgentC
11 if(all members delete AgentB)
12 DELETE_MEMBER (AgentB)
13 if(AgentB ==leader)
14 SEND_MESSAGE ("INFORM","Leader Failed", null, myself)
15 if (I am a leader)
16 SEND_MESSAGE ("INFORM","Failure Report", AgentB, myself)
17 if(RECEIVED_MESSAGE ("REFUTE", AgentB, AgentC))
18 Delete AgentC from suspect_vector(AgentB).
19 if(RECEIVED_MESSAGE ("LEADER", myself, AgentC))
20 mode ← parent
21 SEND_MESSAGE ("INFORM","I am the Leader", myself, group)
22 SEND_MESSAGE ("INFORM","Failure Report", null, myself)

Figure 4. Algorithm for implementing the failure detector

The algorithm of Failure_Detector_Timer is presented in Fig. 5. If a new heartbeat message is not received during timeout, that agent's state is defined as SUSPECT. In this case, an INFORM message is sent to group members. The content of the message contains SUSPECT as the act, the sender's name as the actor, and the suspected agent's name as the argument (lines 6-7 in Fig.5). If agents detect any agent on timeout is suspect, they place the replica in question on their *suspect_vectors*.

```
Failure_Detector_Timer (AgentB, timeout, heartbeat_no))
▷ When it timeouts forAgentB
1  if((heartbeat_no)> heartbeat_data_structure(AgentB))
▷ A new heartbeat is received
2       if(Failure_state (AgentB)=="SUSPECT")
3               SEND_MESSAGE ("INFORM","REFUTE", AgentB, group)
4               Failure_state (AgentB) ⇐ "UNSUSPECT"
5  else
6          Failure_state (AgentB) ⇐ "SUSPECT"
7          SEND_MESSAGE ("INFORM","SUSPECT", AgentB, group)
8          timeout ⇐ timeout+time
```

Figure 5. Algorithm for Failure _Detector_Timer of the failure detector

The agents' dispatcher modules extract information from the incoming messages both the names of suspected members and the names of agents that suspect the members. According to this information, the *suspect_vector* of each member agent is updated. If a member receives a heartbeat message from a suspected member, it then multicasts a message informing that it has refuted a suspected agent (lines 2-3 in Fig. 5).

If members of a group agree that the agent is suspect (line 7 in Figure 4), each agent multicasts a FIPA-ACL message that states the suspected agent will be deleted (line 8 in Fig.4). When a member agent receives DELETE message, it extracts the member's name(s) to be deleted and then updates the *delete_member_vector* with this information. If all members decide to delete the suspected agent, the suspected agent is removed from the *membership_data_structure* and *heartbeat_data_structure* (line 11-12 in Fig. 4).

Since the deleted member is no longer part of the group, it no longer receives multicasting messages. The leader sends a failure report in a FIPA-ACL message to itself in order to activate the Fault Tolerance plan (line 16 in Fig 4). If the leader is deleted from the group, the Leader Election plan is activated in order to select a new leader by sending the *Leader Failed* message (line 13-14 in Fig 4).

B. Implementing the Group Communication Service in the Agency Package

The SEAGENT's communication layer is responsible for abstracting the platform's communication infrastructure implementation and it supports multicasting. The basic multicast operation delivers messages to agents in arbitrary orders. The ordering scheme is implemented by using a sequencer module. Since the dispatcher module is responsible for sending and receiving messages, the sequencer is implemented in the dispatcher module of the Agency package. It assigns a group-specific ascending number to a new message whenever it is received from another agent. The algorithm for the Sequencer is given in Fig. 7, and definitions for the algorithm are given in Fig. 6.

Definitions

Sg: The number of the message assigned by the sequencer

member_no: The number of replica agents in the group.

RECEIVERS: Set of the agents receiving the same messages.

member: member in the membership list.

SEND(message, argument, ORIGINAL_SENDER, receiver): Forward the **ORIGINAL_SENDER** 's **message** containing an **argument** to the **receiver**.

Figure 6. Definitions for the algorithm of the Sequencer module of the group communication service

```
Sequencer
1 At initial Sg ⇐ 0
▷ While multicasting messages
2 for  i ⇐ 0 to the number of members
3       RECEIVERS ⇐ member(i)
4  SEND(message, Sg, ORIGINAL_SENDER, RECEIVERS)
5  Sg ⇐ Sg+1
```

Figure 7. Algorithm for the Sequencer module of the group communication service

C. Implementing Replication Techniques in the Agency Package

In this study, a passive replication technique and a semi-active replication technique (a subtype of active replication) were implemented so that the agents would show non-deterministic behaviors due to their multi-threaded internal structures.

A Semi-active replication technique is employed in SEAGENT as follows: When a FIPA-ACL message containing a request is received by the leader of the replica group, the leader assigns a group specific ascending number to the message and multicasts it to the group. It is then processed by each replica and forwarded to the dispatcher module. If the dispatcher module belongs to the leader, the leader sends the response to the agent waiting for the reply. Therefore, only the leader provides a response to the requesting agent while the actual processing of a request is performed by all replicas. In the absence of failures, replicas process received messages but do not respond, thus their internal state is directly updated by the processing of received messages. If any agent finds out that it has lost the message, it will then get it from the leader.

A passive replication technique is also used to provide fault tolerance when an agent's failure occurs. During a fault-free operation, only the leader processes the requests; the other replicas merely store the sequence of incoming requests. The leader periodically updates its replicas and if a leader fails, one of the replicas will take. When the leader receives a request message, it assigns a number to the message and sends it to the group. In order to update the replicas' states, it serializes the agent's state. It is then written to a file and sent as a byte array to the cloning server of each host, where replicas reside. In the presence of a leader crash, a replica member receives it from the directory and updates its state when selected as a new leader.

Fault tolerance mechanisms are automatically activated when an agent is created. If an agent is created as a fault-tolerant agent, it is created as a parent. The default replication technique is defined as semi-active replication. It can be changed to passive replication by simply sending a FIPA-ACL request at run time.

D. Implementing the Replication Service as Reusable Plans

The replication service creates new replicas, removes agents, and applies static and adaptive policies. If a static fault tolerance policy is applied to a multi-agent system, the replication degree and technique are identified, and the application is initialized. If the adaptive fault tolerance is applied, the replication degree is identified during course of application. However, a group's replication degree and technique can be easily changed by sending a request for a

new replication degree and technique.

1) Fault Tolerance Plan

Whichever policy is applied to an agent system, the leader still needs to receive a failure report to activate the *Fault tolerance* plan. After a suspected replica is deleted from the *membership_data_structure*, a message containing *Failure Report* is sent to the leader by the failure detector. In this case, the leader's goal matcher receiving this message matches it to the *Fault tolerance* goal. To achieve this goal, the *Fault Tolerance* plan outlined in Fig.9 is executed to keep the same replication degree before the failure occurs. Several definitions used in the algorithm of the *Fault Tolerance* plan and the algorithm of this plan are given in Fig.8 and Fig. 9, respectively.

Definitions:
Action no (T_no): name :
no gives the label of **name** action in complex task and it is represented by **(T_no).**
The algorithm is given followed by this.
T_no(Provision): It gives the **provision** of **T_no**.
T_no(Outcome): It gives the **outcome** of **T_no**.
According to the HTN formalism;
T_n<T_m: **T_n** is executed before **T_m**.
(T_n, T_n(Outcome), T_m, T_m(Provision)): The **outcome** of **T_n** is the **provision** of T_m.
(T_n, T_n(Outcome), T_m, T_m(Outcome)): If **T_n** is the subtask of **T_m**, the **outcome** of **T_n** is the **outcome** of **T_m** via outcome disinheritance.
(T_n, T_n(Provision), T_m, T_m(Provision)): If **T_m** is the subtask of **T_n**, the **provision** of **T_n** is the **provision** of **T_m** via provision inheritance.

Figure 8.Definitions for the task networks used in the algorithm of the *Fault Tolerance* plan

PLAN (S_T): FAULT TOLERANCE
(T_1< T_2)Λ (T_1, T_1 (Outcome),T_2, T_2(Provision))Λ(T_2, T_2 (Outcome),S_T,T_2 (Outcome))
1 Action 1 (T_1): Increase the Replication Degree
1.1T_1(Outcome) ← (the replication degree of the group)- (size of the membership_data_structure)
2 Action 2 (T-2): Create New Replicas
2.1T_2(Provision) ← T_1(Outcome)
2.2 for i ← 0 to T-2(Provision)
2.3 SEND_MESSAGE("INFORM","COPY", *null, myself)*
2.4T_2(Outcome) ← "OK"

Figure 9. Algorithm of the *Fault Tolerance* Plan

The task networks consist of complex tasks, and primitive tasks (actions) which may be processed directly. A *'reduction schema'* knowledge defines the decomposition of the complex task to the sub-tasks and the information flow between these sub-tasks and their parent task. The information flow mechanism is as followed: each task represents information acquired by a set of provisions; next, the execution of a task produces outcomes; and lastly, there are links that represent the flow of information between the tasks using these provision and outcome slots. After the task has been completed, an outcome state is produced. The subtasks' provisions and outcomes can be linked to their parent tasks by provision inheritance and outcome disinheritance [13].

The *Fault Tolerance* plan's first task is for the leader to check the *membership_data_structure* to determine how many replicas will be needed to produce the same replication degree. The difference between the size of the *membership_data_structure* and the replication degree of the group equals the number of the replicas needed and is the provision of the *Create New Replicas* task. Thus the link between the Increase the *Replication Degree* task and the *Create New Replicas* task is a provisional link since both tasks construct a non-hierarchical task network.

As soon as the *Create New Replicas* task is assigned a number, it starts to execute the code. In the *Create New Replicas* task, FIPA-ACL messages containing copy requests are prepared according to the number and are then sent to the agent to activate the *Cloning a Replica* plan (implementing the cloning service as a reusable plan). After the copy request messages are received by the leader agent, the *Cloning a Replica* plan is executed.

2) Cloning Service

The cloning service is implemented as a reusable plan structure. The algorithm for the *Cloning a Replica* plan structure is shown in Fig. 10.

PLAN (S_T): CLONING A REPLICA
(T_1< T_2) Λ (T_3, T_3(Outcome),T_4,T_4(Provision)) Λ (T_4,T_4(Outcome), T_5, T_5(Provision)) Λ (T_2,T_2(Provision), T_4, T_4(Provision)) Λ (S_T, S_T (Outcome), T_2,T_2(Outcome)) Λ (T_2, T_2(Outcome), T_5,T_5(Outcome))
1 Action 1 (T_1): Ask a Host for Replication
1.1SEND_MESSAGE("REQUEST","IP FOR COPY", null, AMS)
1.2 T_1(Outcome) ← "OK"
2 Complex Task 1 (T-2): Clone Itself
2.1 T_2(Provision) ← IP
3 Action 2 (T_3): Object Serialization of Agent State
3.1. FILE ← Agent's internal state as a serialized object
3.2 T_3(Outcome)← "OK"
4 Action 3 (T_4): Send Agent's State
4.1T_4(Provision_1) ← IP
4.2T_4(Provision_2) ← Name of FILE
4.3Send FILE via RMI
4.4T_4(Outcome) ← "OK"
5 Action 4 (T_5): Add Agent Identifier to Data Structures
5.1 T_5(Provision) ← New Agent Identifier
5.2 heartbeat_data_structure ← T_5(Provision)
5.3 membership_data_structure ← T_5(Provision)
5.4 T_5(Outcome) ← "OK"

Figure 10. Algorithm of the *Cloning a Replica* plan implementing the cloning service

The plan's first task is to ask AMS to find a suitable host where new replicas will be placed; afterwards, the IP of the host is passed via a provision link and the *Clone Itself* complex task executes the cloning process. In the *Clone Itself* complex task, the cloning server on the remote host where the replicated agent will be placed is contacted by sending messages using RMI. Before sending RMI messages to the cloning server, object serialization of the agent state is performed in the first subtask of the *Clone Itself* complex task. The serialized agent's state is written to a file and sent as a byte array to the cloning server of each host, where each replica resides. In the *Send Agent State* subtask, several messages using RMI are sent to the cloning server at the remote to transfer necessary agent knowledge to perform replication. After sending these RMI messages, this task has outcome state *OK*. In the last subtask of the *Clone Itself* complex task, the Agent Identifier(s) of new replica(s) are added to the *membership_data_structure* and *heartbeat_data_structure*. The *OK* outcome state is propagated upwards via outcome disinheritance.

The cloning server places the unserialized agent's state, the agent's libraries and source code sent in the messages to the paths and then executes the agent's source code. When the replica is initiated, it is registered to AMS and ready to

achieve its fault tolerance goals. Since it has the current version of the leader agent's state, it also contains the last *view* of the group. It multicasts a *JOIN* message informing the other replicas that it has joined the group. The other replicas register the new replica to their *membership_data_structure* and *heartbeat_data_structure* [11].

The replica's only limitation is that it cannot replicate itself. Nevertheless, the replica does have the ability to replicate and respond to other agents, if it is selected as a new leader when the previous leader has crashed.

3) Leader Election

After replicas receive a *Leader Failed* messages from their failure detectors, they execute the *Leader Election* plan to achieve the *Elect a New Leader* goal (see Fig. 11). In the plan's first task, the first member in the membership_data_structure is selected as a new leader as indicated by the *OK* outcome state. The *Assignment of the Leader* task is then enabled via its provision called *AID* (Agent Identifier). During this task, all replicas send an INFORM message to the agent assigned as the leader. The *OK* outcome state is propagated upwards via the outcome disinheritance.

PLAN (S_T): LEADER ELECTION
(T_1< T_2) Λ (T_1, T_1 (Outcome),T_2, T_2(Provision)) Λ
(S_T, S_T (Outcome), T_2,T_2(Outcome))

1 Action 1 (T_1): Ask ID for a New Leader
1.1 Check the membership_data _structure
1.2 T_1(Outcome)←AID of first member of membership_data_structure
2 Action 2 (T_2): Assignment of the Leader
2.1. T_2(Provision)← T_1(Outcome)
2.2 SEND_MESSAGE ("INFORM","LEADER", AID, agent to be assigned as a leader)
2.3 T_2(Outcome)← "OK"

Figure 11. Algorithm of the *Leader Election* plan

When the agent selected to be a leader, its mode is set to parent. It multicasts an *I am the Leader* message to the group and sends a *Failure Report* message to itself to achieve the Fault Tolerance goal (line 19-22 in Fig. 4). Lastly, the *Increase the Replication Degree* and *Create New Replicas* tasks are executed.

4) Adaptive Fault Tolerance Plan

When applying a replication-based fault tolerance policy to a system, a programmer will generally statically define the parameters before the application starts [1], or they will be non-automatically defined at run-time [2]. However, it is very difficult to identify which agent(s) needs to be replicated, and the replication degree in dynamic and/or large scale environments.

In this performance study, the leader agents use an *Adaptive Fault Tolerance* plan (illustrated in Fig. 1) in which an adaptive replication mechanism is performed. The adaptive replication manager sends the agent's criticality (a value between 0 and 1) to each leader agent in the content of a FIPA-ACL message. It indicates the importance of its reliance to a specific agent; therefore, an agent's criticality is important in terms of fault tolerance. Subsequently, if a critical agent fails, other agents that rely on that specific agent will have difficulty in achieving their individual goals. Therefore, critical agents are initialized as fault tolerant agents. Since resources are limited in the environment, they dynamically and automatically share available resources with respect to their criticalities in the environment and they replicate themselves on these resources by applying the adaptive fault tolerance policy. Thus when a critical agent has crashed, their replicas will mask the failure.

When the leader agent receives the message including its criticality from the adaptive replication manager, it executes the *Adaptive Fault Tolerance* plan and gets its criticality value as a provision (outlined in Fig. 12).

PLAN (S_T): ADAPTIVE FAULT-TOLERANCE
(T_1< T_2<T_3) Λ(S_T, S_T(Provision), T_1,T_1(Provision)) Λ (T_2, T_2(Outcome), S_T,S_T(Outcome)) Λ (T_3, T_3(Outcome), S_T,S_T(Outcome)) Λ (T_1, T_1 (Outcome _1),T_2, T_2(Provision))Λ(T_1, T_1 (Outcome _2),T_3, T_3 (Provision))
1. Action 1 (T_1): Increase/Decrease the Replication Degree
1.1T_1(Provision) ⟵ Normalized Criricality of the Agent (W_{ratio})
1.2 if(Adaptive Fault Tolerance Policy==Adaptive)
1.3 Rdegree ⟵ CALCULATE_NUMBER_OF_REPLICAS (T_1(Provision)) using (Eq. 1)
1.4 if(Rdegree — No_R > 0) T_1(Outcome_1) ⟵ Increase
else T_1(Outcome_2) ⟵ Decrease
2. Action 2 (T_2): Create New Replicas
2.1T_2(Provision) ⟵ (Rdegree-No_R)
2.2for i ⟵ 0 to T_2(Provision)
2.3 SEND_MESSAGE("INFORM", "COPY", null, myself)
2.4 T_2(Outcome) ⟵ "OK"
3 Action 3 (T_3): Decrease the Replication Degree
3.1T_3(Provision) ⟵ (No_R-Rdegree)
3.2 for i ⟵ 0 to T_3(Provision)
3.3 SEND_MESSAGE ("INFORM", "DECREASE THE REPLICATION DEGREE", null, myself)
3.4T_3(Outcome) ⟵ "OK"

Figure 12. Algorithm of the *Adaptive Fault Tolerance* plan

The first task of the *Adaptive Fault Tolerance* plan (*Increase/Decrease the Replication Degree*) is to determine the group's replication degree by using criticality of the agent (*Wratio)* sent by the adaptive replication manager. Due to limitation of resources, the new replication degree for each leader agent is defined as follows:

$$R\deg ree = rounded(W_{ratio} \times R\_\max) \quad (1)$$

$$No\_agent = R\deg ree - No\_R \quad (2)$$

Rdegree conveys the replication degree in the current sampling period and *R_max* identifies the number of available resources that define the maximum number of possible simultaneous replicas. Lastly, *No_R* gives the number of replicas in the group.

If No_agent is positive, then the outcome state is expressed as *Increase*. When the *Create New Replicas* task identifies the value of $(R\deg ree - No\_R)$ as its provision, it executes its code. In the *Create New Replicas* task, as many FIPA-ACL messages containing copy request are prepared as the value of $(R\deg ree - No\_R)$ and are then sent to the agent itself in order to activate the Cloning a Replica plan. After the COPY request messages are received by the leader agent, the Cloning a Replica plan is executed. New replicas are created; therefore, the replication degree of the group reaches to the value of $R\deg ree$.

If No_agent is negative, then the outcome state is expressed as *Decrease*. When the *Decrease the Replication Degree* task identifies the value of $(No\_R - R\deg ree)$ as its provision, it executes its code to decrease the replication degree of the group. In the *Decrease the Replication Degree* task, as many FIPA-ACL messages containing the request to decrease the replication degree of the group are prepared as

the value of $(No\_R-R\deg ree)$ and are then sent to the agent itself in order to activate the *Decreasing Replication Degree* plan. After the leader receives these messages, it executes the *Decrease the Replication Degree* plan.

In the first subtask of this plan, a FIPA message is sent to AMS to determine which replica is performing the worst. After AMS identifies the agent that is to be removed, the next subtask is executed which kills the agent and the information related to this agent is deleted from the *membership_data_structure*.

E. Observation Service and Calculate Criticalities of Agents Plan

Monitoring is necessary in order to acquire specific information that determines the criticality of agents. The information is acquired from either the system-level such as communication load and processing time etc. or the application level information such as the importance of messages and the role criticalities of the agents [3], [7], [11], and [24-25].

In order to collect and evaluate criticality related data, a *replication manager* role is proposed. The agent that enacts the role of the *replication manager* is called the replication manager agent. It is a centralized agent that controls leader agents in the MAS during runtime and is the single point of failure. In order to prevent the replication manager from failing, it also needs to be replicated (see Fig. 1).

The replication manager stores the current states of the leader agents. It receives criticality related messages sent by the leader agents and then forwards extracted data to the *observation service* plan. All data received from leaders is stored in a data structure that is updated periodically. Afterwards, the criticalities of agents are calculated by using the collected data in the *Calculate Criticalities of Agents* plan (illustrated in Fig. 1). How to calculate criticalities of agents is explained in [7], [11], [24]. Criticality values of leader agents are sent to the leader agents and used in the adaptive fault tolerance plan that is implemented as a reusable plan using HTN formalism.

In order to better illustrate how fault-tolerant services are implemented and integrated into the SEAGENT platform, a performance analysis of the proposed approach is presented in the following section.

IV. A PERFORMANCE ANALYSIS OF THE PROPOSED IMPLEMENTATION APPROACH

The fault tolerance approach presented in this paper has been implemented within SEAGENT's internal architecture. In order to evaluate the presented approach, an agent system was designed that included library assistant agents and user agents that were uniquely designed to query library assistant agents. Each library assistant agent monitors a different library and has the library knowledge using the library ontology. In the case study, each user agent directly sent a book request to all of the library assistant agents. The library assistant agent initiated only one plan to match the request to the book ontology instance(s) and return the matched books' descriptions within a FIPA-ACL message. When the user agent received responses from the library assistant agents, it selected a library based on the responses and presented the result to the user. Each library assistant agent was a critical agent for the system's operation; therefore, it was initialized as a fault-tolerant agent [11].

The agent system was implemented in the SEAGENT platform and Java Version 1.5.0. The tests were performed on a computer with Intel Core2 Q6600 CPU and 2GB of RAM.

The evaluation consisted of four tests: the cost resulting from agent replication as the number of requests increased; the cost resulting from agent replication as the number of replicas increased; the cost of adding new replicas; and lastly the cost of failure recovery in the case of a crash of a replica and/or leader. In the next sections, both the tests and their results are presented in detail.

A. The Cost of Replication

In order to evaluate the cost of replication, the response times of a replicated group employing the semi-active replication technique were observed. As the number of requests sent to the group increased, the number of replicas also increased. Therefore, a test environment was implemented, which included a library assistant agent leader and its replicas in the number range from 5 to 20, and a user agent that queried the library assistant agent. In order to report the effect of the number of replicas to the response time of the system, the user agent sent requests to the leader and the response times for queries were measured. The response time was calculated by measuring the amount of time taken for the user agent to receive the reply from a leader agent after sending its request to the leader. The results of the first tests are illustrated in Fig. 13.

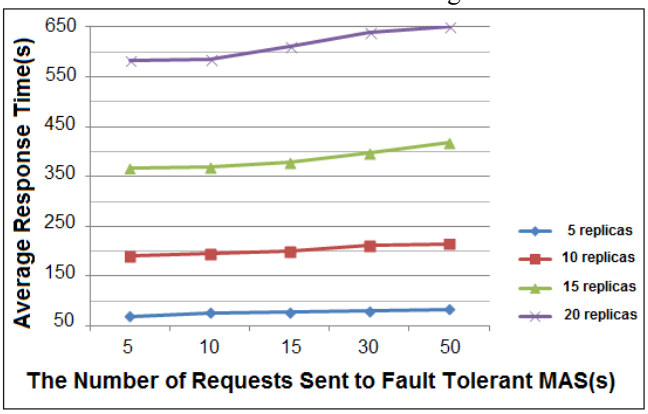

Figure 13. Evaluation of the cost of replication as the number of requests sent to the replica groups increases

As indicated by the graphs, the average response times of the system applying the semi-active replication increased with the number of requests sent in a group. The increase in response time was expected, since the number of requests sent to the system increased. The leader of the group multicasts all incoming requests to the replicas and all replicas process these requests. Moreover, as the number of replicas increased, the response time of the system increased, as seen from Fig. 13.

In SEAGENT environment, communication module uses the RMI based communication infrastructure and all functionalities of internal architecture are based on threads and implemented as separate modules. Therefore, when all agents were created in a single machine, then agents' threads initialized. In addition, the number of the messages exchanged increased with the number of agents due to the multicasting of request and heartbeat messages.

In the second part of the test, the response times of a replicated group were observed, as the number of replicas in the group increased when the number of requests was set to 40. In this test, a library assistant agent leader has replicas in the number range from 5 to 50. In order to determine the effect of the number of replicas to the response time of the system, the user agent sent queries to the leader and the response times of the queries were measured in this test.

The results of the second test are illustrated in Fig. 14. As indicated by the graph, the average response times of the system increased exponentially with the number of replicas in a group. The increase in response time was expected, since all replicas process client requests. Moreover, the number of the messages exchanged increased with the number of agents due to the multicasting of requests and heartbeat messages.

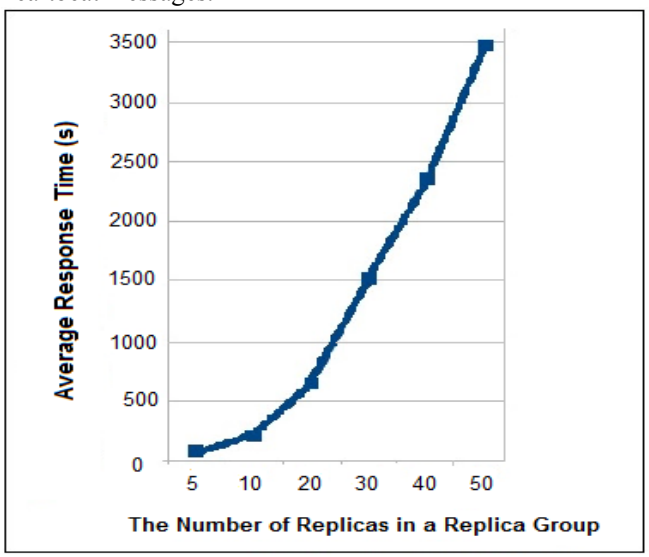

Figure 14. Evaluation of the cost of replication for 40 requests as the number of replicas in the group increases

In replication-based approaches, there are multiple replicas of the same agent that run concurrently. The cost of the replication of an agent is the sum of the cost of replica creation, replica usage, and overheads incurred by the coordination of the replicas. While applying static fault tolerance, a certain number of resources in a system are reserved to provide fault tolerance to multi-agent systems. Changing the replication degree may decrease the replication cost of an agent as illustrated in Fig. 13 and Fig. 14. Thus an adaptive fault tolerance policy enables a system to automatically change its replication degree in accordance to its environment. Findings of the cost of applying the adaptive fault tolerance policy were presented in previous works [3], [11], [24]. However, the cost resulting from monitoring of the environment and the agents' behaviors is inevitable while determining the agents' criticalities and adapting replica groups in an adaptive fault tolerance policy.

B. Evaluation of the Cloning Service

The results of the third test are illustrated in Fig. 15. The response time is the time it took a leader to receive heartbeat messages from new replicas after receiving COPY requests for replicating new replicas. As the number of replicas to be replicated increased, the amount of time taken to replicate new replicas also linearly increased.

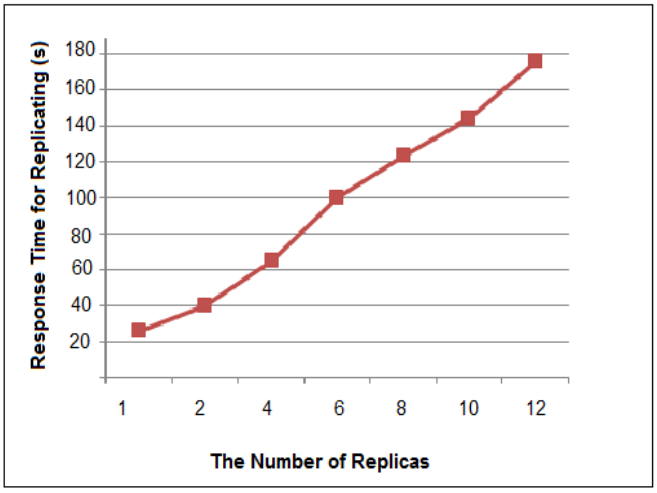

Figure 15. Evaluation of the cost of adding new replicas and the cloning service

C. Evaluation of Failure Occurrence

1) Crash of a Replica:

In order to observe a replica crash, the replication technique was set as semi-active replication and the replication degree was set as 5 in the system. Next, a failure simulator sent a kill message to one of the replicas. The agent receiving the kill message eventually stopped its threads. The group members' failure detector mechanisms detected that one of the replicas had crashed and, removed it from their *membership_data_structures*. As a result, the leader sent a Failure Report message to itself. When the leader received the Failure Report message, it matched the message to the Fault Tolerance plan. The *Increase the Replication Degree* and *Create New Replicas* actions were executed to replicate a new member in the same host. After receiving the message from failure detector, all operations were executed in a time frame ranging from 30146-32348 ms.

2) Crash of a Leader:

Also, the system was observed when a leader crashed. In order to simulate the presence of failures, the failure simulator sent a *kill* message to the leader, whereby it eventually stopped its threads. Upon receiving the *Leader Failed* messages from their failure detectors, they executed the *Leader Election* plan in order to achieve the *Election of a new Leader* goal. According to the plan, the first member in the *membership_data_structure* was selected as the new leader. The new leader then sent a *Failure Report* message to itself to achieve the *Fault Tolerance* goal by executing the *Fault Tolerance* plan. After receiving the message from failure detector, all operations were executed in a time frame ranging from 44605-47853ms.

Moreover, it was also possible to evaluate the change of the *Leader Election* plan at run-time. Two plan structures were designed for leader election. In the first plan, replicas asked AMS to select a new leader in the *Ask ID for a New Leader* task. After receiving the new leader's AID from AMS, all replicas sent an INFORM message to the agent assigned in the *Assignment of the Leader* task. In the second plan, the first member in the *membership_data_structure* was selected as a new leader. It was observed that the change of plan structure at run time was possible by implementing the following scenario. While the leader and

its replicas were processing the incoming requests, the leader was killed. When the replicas received the *Leader Failed* messages from their failure detectors, they executed the *Leader Election* plan. According to the first plan, a new leader selected by AMS was identified. In the second part, one of the user agents sent a FIPA-ACL request message to the leader to change the *Leader Election* plan. After the leader received that message, it was killed by sending a kill message. After the replicas executed the *Leader Election* plan, the leader which was the first agent in the *membership_data_structure* was selected as the new leader. Thus, it was observed that the *Leader Election* plan (any reusable plan structure in this approach) could be changed by sending a request at runtime.

V. CONCLUSION

In this paper, an approach to provide replication-based fault tolerance to goal-oriented multi-agent systems was presented. In this approach, new goals coming from fault tolerance requirements, the plans of these goals, and reusable fault tolerance services that could be used by the plans of these goals were identified. Implementations of fault tolerance services were elaborated and integration of these services to a goal-oriented architecture was explained.

The approach provided flexibility to multi-agent organizations in terms of fault tolerance because the fault tolerance policies were implemented as reusable plan structures. Thus, whenever an agent needed to be made fault-tolerant, the action was performed by sending a request to that agent. Moreover, this approach was flexible due the fact that it was easy to modify existing plans, remove unnecessary parts from an existing plan, or create an entire new plan.

REFERENCES

[1] A. Fedoruk and R. Deters, "Improving fault-tolerance by replicating agents," Proceedings of First International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna, Italy, 2002.

[2] J. Ren, M. Cukier, P. Rubel, W. Sanders, D. Karr," Building dependable distributed applications using AQuA," Proceeding of the 4th IEEE International Symp. On High Assurance Systems Engineering, 1999, pp. 189-196.

[3] Z. Guessoum, M. Ziane, and N. Faci, "Monitoring and organizational-level adaptation of multi-agent systems," Third International Joint Conference on Autonomous Agents - AAMAS'04, ACM, New York City, 2004, pp. 514-522.

[4] S. Kumar, P. Cohen, and H. J. Levesque, " The adaptive agent architecture: Achieving fault-tolerance using persistent broker teams," Proceedings of the Fourth International Conference on Multi-Agent Systems, 2000.

[5] Klein and C. Dallarocas, "Exception handling in agent systems," O. Etzioni, J. P. Muller. and J. M. Bradshaw editors, Proceedings of the Third International Conference on Agents (Agents'99), Seattle, WA, 1999, pp. 62-68.

[6] S. Hagg, "A sentinel approach to fault handling in multi-agent systems," Proceedings of the second Australian Workshop on Distributed AI, in conjunction with the Fourth Pacific Rim International Conference on Artificial Intelligence (PRICAI'96), Cairns, Australia, 1996.

[7] Z. Guessoum , J.-P. Briot, Z. Charpentier, S. Aknine, O. Marin, and P. Sens, "Dynamic and adaptive replication for large-scale reliable multi-agent systems," Proc. ICSE'02 First International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'02), ACM. Orlando FL, U.S.A, 2002.

[8] Z. Guessoum, and J.–P. Briot, "From active objects to autonomous agents," IEEE Concurrency, 7(3), pp. 68-76, 1999.

[9] N. Faci, Z. Guessoum, and O. Marin, "DimaX: A fault tolerant multi-agent platform," Proc. ICSE'06 Fifth International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'06), ACM. Shangai, China, 2006.

[10] K. Erol, J. Hendler, D. S. Nau, R. Tsuneto, "A critical look at critics in HTN planning," Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), 1995, pp. 1592-1598.

[11] S. Bora and O. Dikenelli, "Implementing a multi agent organization that changes its fault tolerance policy at run-time," Proceedings of ESAW'05, Lecture Notes in Computer Science, Berlin, Germany, Department of Computer Engineering Ege University, Springer Verlag, 2006.

[12] O. Dikeneli et al., "SEAGENT: A platform for developing semantic web based multi agent systems," Fourth International Joint Conference on Autonomous Agents - AAMAS05, 2005.

[13] O. Gurcan, G. Kardas, O. Gumus, E. E. Ekinci, and O. Dikenelli, "A planner for implementing semantic service agents based on semantic web services iniative architecture," The Workshop on Service-Oriented Computing and Agent-based Engineering (SOCABE'06), 2006.

[14] D. Powell, "Delta-4: A generic architecture for dependable distributed computing," in Springer Verlag, 1991.

[15] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," ACM Computing Surveys 33(4), pp. 1-43, 2001.

[16] F. Cristian, "Fault-tolerance in the Advanced Automation System," 20th International Conference on Fault-Tolerant Computing, Newcastle upon Tyne, England, 1990.

[17] S. Mishra, Consul: A Communication Substrate for Fault-tolerant Distributed Programs. PhD thesis, Dept. of Computer Science, Univ. of Arizona, Tuscon, Arizona 1992.

[18] N. Elmootazbellah and W. Zwaenepoel, "Replicated distributed processes," Proceedings of the Twenty-Second International Symposium on Fault Tolerant Computing (FTCS-22), 1992, pp. 18-27.

[19] M. Stollberg and F. Rhomberg, Survey on Goal-driven Architectures. Technical Report, 2006.

[20] M. Paolucci, D. Kalp, A. Pannu, O. Shehory, and K. Sycara , "A planning component for RETSINA agents," Intelligent Agents VI, LNAI 1757, N. R. Jennings and Y. Lesperance, eds., Springer Verlag, 2000.

[21] O. Shehory, K. Sycara, P. Chalasani, and S. Jha, "Agent cloning: An approach to agent mobility and resource allocation," IEEE Communications, Vol. 36, No. 7, pp. 58-67, 1998.

[22] K. Decker, K. Sycara, and M. Williamson, "Cloning for intelligent adaptive information agents," The Second Australian Workshop on Distributed Artificial Intelligence, Lecture Notes in Computer Science, Springer Verlag, 1996, pp. 63-75,

[23] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," Journal of the ACM, Vol.43, No.2, pp. 225-267, 1996.

[24] S. Bora, and O. Dikenelli, "Applying feedback control in adaptive replication in fault tolerant multi-agent organizations," Proc. ICSE'06

[25] Fifth International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'06), ACM. Shangai, China, 2006.

[26] S. Bora and O. Dikenelli, "Experience with feedback control mechanisms in self-replicating multi-agent systems," Proceedings of 5th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'07), Lecture Notes in Computer Science, Springer Verlag, 2007, pp. 133-142.