# AN EMOTIONAL DECISION MAKING HELP PROVISION APPROACH TO DISTRIBUTED FAULT TOLERANCE IN MAS

**MOHSEN LESANI (m.lesani@ece.ut.ac.ir)**
**AMIR MOGHIMI (a.moghimi@ece.ut.ac.ir)**
**ALI AKHAVAN BITAGHSIR (a.akhavan@ece.ut.ac.ir)**
**CARO LUCAS (lucas@ipm.ir)**
**ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT, FACULTY OF ENGINEERING, UNIVERSITY OF TEHRAN, IRAN**

## ABSTRACT

Fault is inevitable especially in MASs (multi-agent system) because of their distributed nature. This paper introduces a new approach for fault tolerance using help provision and emotional decision-making. Tasks that are split into criticality-assigned real-time subtasks according to their precedence graph are distributed among specialized agents with different skills. If a fault occurs for an agent in a way that it cannot continue its task, the agent requests help from the others with the same skill to redo or continue its task. Requested agents would help the faulty agent based on their nervousness on their own tasks compared to his task. It is also possible for an agent to discover death of another agent, which has accepted one of his tasks, by polling. An implementation using JADE platform is presented in this paper and the results are reported.

**KEYWORDS:** Distributed Fault Tolerance, Multi Agent Systems, Help Provision, Emotion.

## 1. INTRODUCTION

Fault is probable in every system. It happens when a system's component does not work or respond the way it is expected to. The fault should be detected gracefully in the system, isolated and resolved properly to prevent it from propagating through the whole system and resulting in a system failure. Such a system is called fault tolerant. Fault tolerance is done both in hardware and software. Main idea behind general fault tolerance hardware methods is duplication with comparison. NMR (N-Modular Redundancy) maintains the system with n duplicates of a module, such as TMR (Triple Modular Redundancy) with a voter that spots the fault in one of the three duplicated modules with comparison. Standby sparing is used to substitute a spare module for a same module that is known to be faulty. Standby sparing can be of two types of hot standby sparing and cold standby sparing. One way to run away from the cost of modular redundancy is through Time Redundancy that is to perform the task two times with the same module. Simple examples of software fault-tolerance methods are consistency checks such as to perform a limited number of tests on a hardware or software module with known results or capability check of the system's RAM in boot up time through a series of write and read instructions. Other methods include N-version programming and Recovery block.

Multi-agent systems (MAS) are well suited for developing complex distributed systems and also are prone to failures. Traditionally, to have a fault tolerant system, we can build subsystems from redundant components placed in parallel so if one fails the other can take over. This strategy appears as agent replication in MAS paradigm. It relies on the fact that all the agents will not fault the same time, if one faults others take its place or resolve its fault by result comparison.

Agent replication has the disadvantage of wasting system resources and also in real world applications it is rare to have an agent group dedicated for when an agent faults; but agents are assigned different tasks to make best use of their computational power. This paper introduces some fault tolerance techniques that provide help for faulty agents in order to reconfigure the

system. In this new method, there is no extra or central agent, sentinel or broker to observe the agents and redistribute the tasks among the agents to clear the fault. This method is more coherent with the definition of an agent as an autonomous entity. The help strategy is presented and the implementation results in an MAS environment are discussed. The main idea is that the agents can decide about their best actions in the fault situation. As a result, the total system can be considered as a well-designed intelligent system. In a non-deterministic system, an agent should consider different possibilities, then make the most appropriate decision based on a non-deterministic method and perform the most suitable action.

## 2. EMOTIONAL DECISION-MAKING BASED HELP PROVISION

Mirian [1,2,3] used help provision approach, based on OS-like scheduling strategies for task selection. In this section, after explaining the general assumptions taken, a detailed description of our method that is based on emotional decision-making is presented. In this paper, it is assumed that we have some agents in a society. Each of these agents has some tasks that should be done. These tasks can be parts of a larger task that is divided into some subtasks. How to decompose the task to these subtasks is another hard problem in distributed systems and is out of the scope of this paper. Here it is assumed that the task is divided into some real time subtasks that have no dependencies and the process time is approximated for each of them.

Different tasks need different skills. Not all the agents have all the skills. So there is a need for identifying the agents with a special skill and also a kind of interaction protocol for requesting a task and getting back the result. The need for tasks to be done emerges gradually in each agent of the society. Tasks emerge with predefined criticalities, deadlines and approximated process times and also they may arrive simultaneously. Every arrived task needs a special skill, so the agents with this skill are searched for. Then the task is advertised to all of the skilled agents. The requested agents answer to the request according to their emotional state. If they are not busy they agree to accept the task for sure, otherwise their nervousness on their own task is compared to their nervousness to the requested task and if the latter is higher they agree. When an agent agrees he sends his nervousness on his own task to the requesting agent. Nervousness is computed as follows:

$$\text{Nervousness} = \text{Criticality} * \alpha / \text{Remained Time} \quad (1)$$
$$\text{Remained Time} = \text{Approximated Process Time} - \text{Done Process Time}$$

More critical a task is, the agent is more nervous on doing it; so in the above equation the nervousness is proportional to task criticality. Also if there are just a few steps to complete a task, the agent is more nervous not to lose it. Hence it appears in the denominator of the equation. $\alpha$ is a constant factor to magnify the nervousness value. The agent waits an amount of time for the incoming agree messages. So the agent may receive more than one agree message for a task and he should decide which agent to offer the task to. If the agent waits longer, it may receive more agreeing messages and it is more probable to decide better. On the other hand, waiting longer leaves less time for fault situations. When a fault occurs there is a need of some extra time to request help and retransmit the task state to a healthy agent. The waiting time is computed as below:

$$\text{Waiting Time} = \text{Spare Time} / \text{Fault Rate} \quad (2)$$
$$\text{Spare Time} = \text{Deadline} - \text{Current Time} - \text{Approximated Process Time}$$

The more the spare time is, the more we can wait for other agreeing agents. Fault rate is a parameter that approximates the degree of fault in the environment. If it is 1 then the environment never faults so the agent waits the whole spare time. If the fault rate is higher the agent waits less to save more time for the probable future faults.

To reach the overall goal of the system, which is doing the most critical tasks, the agent would send the task to the agreeing agent with the minimum nervousness. The performance of the system is defined as:

$$Performance = \frac{1}{\sum_{j=1}^{M}\sum_{i=1}^{N} C_{ij}} \sum_{j=1}^{M}\sum_{i=1}^{n} C_{ij}$$

Where n is the number of successfully performed tasks with $C_i$ criticality while N is the number of assigned tasks with this criticality coefficient. M is the number of available different criticality classes. The minimum nervousness selection causes the more critical tasks to have more priority. Although if a task is about to finish and it can increase the overall performance of the system, it is prior. This also improves the system total free time, which is an important factor in future fault tolerance ability of the system.

The agent rejects a task that it has agreed to do when it has accepted a new task from another agent in the time that the requesting agent was waiting for other agent responses, if it is more nervous on the new task rather than the offered task. It may also reject if the time passed has made it more nervous now on the task it was and is processing. If an agent gets a reject message, he will try offering the task to another agreeing agent or will advertise the task again if all of the previously agreeing agents rejected him. At worst the task is advertised repeatedly till its deadline. It is also possible that the processing agent drops the accepted task if it receives a task that is more nervous on from another agent.

Two kinds of fault are possible: delayed fault and fatal fault. Delayed fault is a fault that has a reprieve before it happens, so the agent has a short time to request help. In the case of a fatal fault, the agent shuts down unexpectedly and has no time to request help. To tolerate fatal faults a polling mechanism is embodied into the agents so that the requesting agent polls the processing agent periodically. The polling period is determined according to the following equation:

$$\text{Poll Period} = \text{Approximated Process Time} / \text{Fault Rate} \qquad (3)$$

If the fault rate is high the requesting agent will poll more because it is more probable for the processing agent to fault. If the requesting agent notices a fatal fault by not getting a poll acknowledge for a predefined number of times, it advertises his task again to be restarted.

If a delayed fault occurs the faulty agent recognizes his forthcoming fault so he requests help from healthy agents with the same skill and as he is in a critical situation he sends the task state to the first agreeing agent. The agreeing agent has agreed based on the nervousness computation and also may retain his own task if its deadline is sufficiently far. The helping agent continues the task from the received task state. The faulty agent also informs the primary requesting agent that it cannot continue his task anymore. The helping agent gets the primary requesting agent address from the faulty agent and introduces himself so that he will poll him from now on.

## 3. IMPLEMENTATION

Implementation of the described approach is done in a software platform named JADE (**J**ava **A**gent **DE**velopment Framework). **JADE** is a software framework fully implemented in Java language. It simplifies implementation of multi-agent systems through a middle-ware that claims to comply with the FIPA specifications and through a set of tools that supports the debugging and deployment phase [4]. FIPA is a non-profit organization aimed at producing standards for the interoperation of heterogeneous software agents.

All agent types in JADE platform are Java classes inherited from a common class named Agent that provides the basic functionality for an agent to come into life and perform its life cycle

operations. In this work, a class named *FTAgent* (Fault Tolerant Agent) is implemented which registers itself and its corresponding skills in DF when it setups. It receives emerged tasks as time passes by and advertises them to the capable agents. It also answers the requests of other agents and processes their tasks. Tasks arrive at unknown times. A task is an instance of the Task class. Task class contains the task id, required skill, criticality, deadline and task data.

In JADE framework, every agent has some behaviors implemented as subclasses of *Behavior* class. *Behavior* is an abstraction of some actions performed by the agent autonomously. There are two main behaviors implemented in *FTAgent* named *TaskAdvertiserBehaviour* and *TaskReceiverBehaviour*. *TaskAdvertiserBehaviour* is responsible for advertising emerged tasks to the agents that have the required skills. These agents are retrieved from DF. This behavior implements the initiator part of the *TaskAllocationProtocol*, an interaction protocol designed for advertising a task. *TaskReceiverBehaviour* implements the responder part of this protocol and listens for the incoming requests of tasks to be done. The representation of this IP (Interaction Protocol) is given in Figure 1 by an extension to UML sequence diagram. This diagram shows three different interaction cases that may happen in the protocol between an initiator and a responder not one interaction between one initiator and three responders. Polling mechanism is implemented through an interaction protocol named *PollingProtocol.*Initiator part of it is implemented in *TaskAdvertiserBehaviour* and responder part in *TaskReceiverBehaviour*.

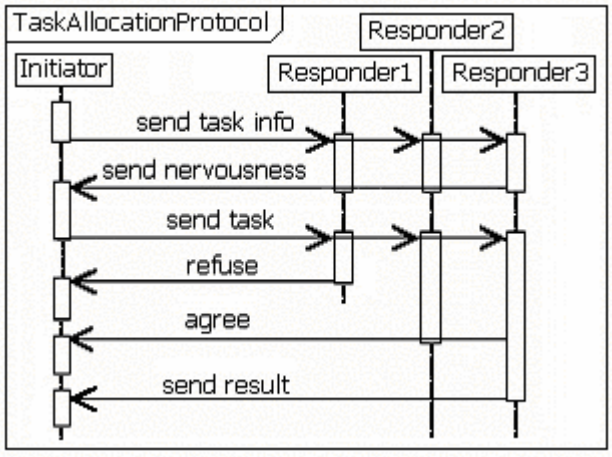


*Figure 1. Task allocation protocol sequence diagram.*

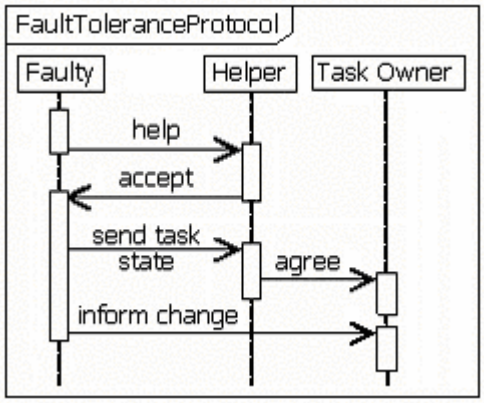


*Figure 2. Fault tolerance protocol sequence diagram.*

When *TaskReceiverBehaviour* accepts a task from another agent it starts another behavior named *TaskProcessorBehaviour* which is responsible for doing the task and sending the results back. When a fatal fault occurs, the agent dies unexpectedly. So its entire behaviors, including *TaskProcessorBehaviour*, stop working that means its current task is dropped. The part of *TaskReceiverBehaviour* that replies polls stops, so the polling agent discovers the fatal fault.

In the case of delayed fault, the agent understands that it is going to fault in a short amount of time. So the *TaskProcessorBehaviour* saves the current task state and stops processing the task. *TaskReceiverBehaviour* ignores any incoming messages except poll messages. *TaskAdvertiserBehaviour* does not advertise emerged tasks any more; rather it requests help from other healthy agents by advertising the agent's current task. It also accepts the results of its tasks done by other agents. The part of the *TaskAdvertiserBehaviour* that polls other agents stops too.

The way an agent decides to help another one is exactly similar to accepting a new emerged task from an agent. In the delayed fault situation, the faulty agent does not wait for some extra agents to send accept; it gives the task state saved as a *TaskState* class instance to the first agent that sends accept. It also notifies the *TaskAdvertiserBehaviour* of task owner that the helping

agent now continues its task processing. So task owner will poll that agent from now on rather than the faulty agent. This prevents the task owner to think that a fatal fault has occurred and advertise the task again. This interaction protocol named *FaultToleranceProtocol* is shown in Figure 2. An agent may recover from fault, which all of its behaviors are restarted and it is ready for any further activities.

## 4. SIMULATION

For simulating fault situations, two fault models are considered: static and dynamic. For static model, *StaticFaultGeneratorBehaviour* is implemented which reads static faults from an XML file and generates them as is specified. Every fault has the parameters delay, start time and finish time. Delay is time that the agent is aware of its future fault before it happens. When delay is zero for a fault, it will be fatal and when it is a positive number it will be a delayed fault.

*DynamicFaultGeneratorBehaviour* reads dynamic fault model parameters that are probability, period, delay and duration from the XML file. This model is periodic where in each period the agent may fault with the specified probability. If a fault is going to occur, it has the specified delay and duration. If delay is zero, all faults will be fatal.

Two trials simulated are presented in Figure 3 where the skills, tasks and faults of the agents are as given in Table 1. All times are presented in seconds with precision of milliseconds. Values used for parameters of the algorithm in this simulation are $\alpha$ = 100,000 and FaultRate = 10. This simulation is done on a PC with an Intel Pentium III 750MHz processor and 224MB of RAM running Microsoft Windows XP Professional. Simulation is done using JADE 3.0b1. Four cases, named A to D, are considered in these tests.

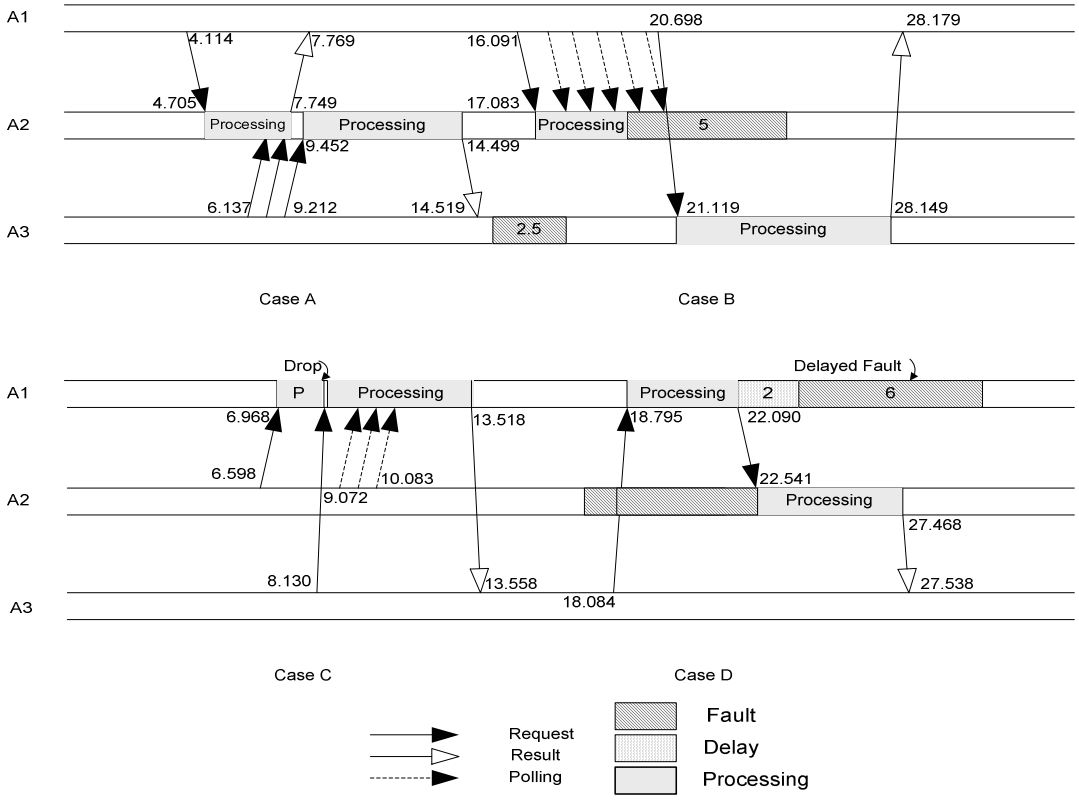


*Figure 3. Simulated trials time diagram.*

| Agent | Skills | Tasks<br>(id, skill, arrival time, process time, deadline, criticality) | Faults<br>(start, length, delay) |
|---|---|---|---|
| A1 | B, D | (1, A, 4, 3, 10, 10), (5, C, 16, 7, 30, 5) | (24, 30, 2) |
| A2 | A, C, D | (3, B, 6.5, 5, 13, 3) | (18, 23, 0)<br>(17, 21.5, 0) |
| A3 | C | (2, A, 6, 5, 15, 6), (4, B, 8, 5, 15, 10)<br>(6, D, 18, 8, 30, 5) | (15, 17.5, 0) |

Table 1. Agents and tasks specifications of simulated trials.

Table 2 shows simulation results of two other scenarios with and without methods introduced in this paper. It is realized that the performance is significantly better in the proposed method in comparison to the other two approaches. Also the effect of considering emotion in decision-making is clearly seen in the table.

| | # of Agents | # of Tasks | Proposed Method | | Emotional Task Allocation without FT* | | Simple Task Allocation without FT | | Max Criticality |
|---|---|---|---|---|---|---|---|---|---|
| | | | C† | P‡ | C | P | C | P | |
| Scenario 1 | 3 | 5 | 25 | 89.2% | 20 | 71.4% | 13 | 46.4% | 28 |
| Scenario 2 | 3 | 5 | 27 | 96.4% | 20 | 66.6% | 13 | 43.3% | 30 |

Table 2. Simulation results of two other scenarios.

## 5. CONCLUSIONS AND FUTURE WORKS

These experiments show that this algorithm works well with limited number of agents and tasks. Larger test cases with more number of agents in different faults environments are needed to show how this algorithm behaves in more complex situations. In the presented approach, because agents find each other using DF, the DF introduces a central point of failure into the system. The solution to centralized DF problem relies on distributed management of the openness. Tolerating this kind of fault and considering communication time precisely are the next steps in this work.

## 6. REFERENCES


[1] M. Mirian et al, "A fault tolerant multi-agent system with non-deterministic decision-making for task allocation," Proc. ERSA, Las Vegas, Nevada, USA 2003, pp. 312-315.

[2] M. Mirian et al, "Agent's role reconfiguration based on decision-making for fault-recovering in multi-agent systems," Workshop of EURASIA-ICT, Shiraz, Iran 2002, pp. 289-293.

[3] M. Mirian et al, "A new task redistribution method for fault clearing in multi-agent systems," Proc. IEEE Int. Conf. on Systems, Man and Cybernetics, Vol. 2, Hammamet, Tunisia 2002.

[4] JADE Administrator and Programmer's guide, see *http://sharon.cselt.it/projects/jade/* for more information.

[5] L. Vercouter, "A fault-tolerant open MAS," Proc. The First International Joint Conference on Autonomous Agents and Multiagent Systems, 2002, part 2, pp. 670-671.

[6] M. Wooldridge et al, "The Gaia methodology for agent-oriented analysis and design," *Journal of Autonomous Agents and Multi-Agent Systems*, Vol. 3, 2000, pp. 285-312.

[7] Z. Guessou et al. "A fault-tolerant multi-agent framework," Proc. The First International Joint Conference on Autonomous Agents and Multiagent Systems, 2002, part 2, pp. 672-673.


* Fault Tolerance
† Criticality
‡ Performance