Proceedings of the International Multiconference on Computer Science and Information Technology pp. 217–222
ISBN 978-83-60810-27-9
ISSN 1896-7094

Improving Fault-Tolerance of Distributed Multi-Agent Systems with Mobile Network-Management Agents

Dejan Mitrović
Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Serbia
Email: dejan@dmi.uns.ac.rs

Zoran Budimac, Mirjana Ivanović
Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Serbia
Email: {zjb, mira}@dmi.uns.ac.rs

Milan Vidaković
Faculty of Technical Sciences, University of Novi Sad, Serbia
Email: minja@uns.ac.rs

***Abstract*—Large-scale agent-based software solutions need to be able to assure constant delivery of services to end-users, regardless of the underlying software or hardware failures. Fault-tolerance of multi-agent systems is, therefore, an important issue. We present an easy and flexible way of introducing fault-tolerance to existing agent frameworks. The approach is based on two new types of mobile agents that manage efficient construction and maintenance of fault-tolerant multi-agent system networks, and implement a robust agent tracking technique.**

I. Introduction

AGENT technology represents one of the most consistent approaches to distributed systems development. *Software agents* can be defined as executable software entities with varying degrees of intelligence, autonomy, and the ability to communicate to each other in order to solve common problems. An important feature of some software agents is *mobility*, which enables them to move from one node in a network to another. Migration path is often chosen autonomously by the agent, and in general case it cannot be predicted.

Agents need an environment in which they can execute their tasks. A *multi-agent system* (MAS), or *agent framework*, represents a programming environment that controls agent's life-cycle and provides it with all the necessary mechanisms for task execution.

Multi-agent systems are applicable to a wide range of problems, such as information retrieval [11], mobile telecommunication networks [7], and power supply management [20], [21]. These solutions often employ physically distributed, interlinked agent frameworks, in combination with migrating agents. When compared to the standard client-server model, agent-based approach yields in improved performance and flexibility, reduced bandwidth requirements and, ultimately, cost.

Large-scale agent-based solutions are, as any software system, exposed to failures. Communication interruptions in systems comprising a large number of frameworks can occur due to traffic overload or hardware failures; agents carrying important gathered data might become lost because of a software glitch. These faults, however, must not affect the functioning of the system as a whole – power supply must not be interrupted because of a failed MAS node in the network. Fault-tolerance mechanisms should, therefore, be an important part of any MAS implementation.

Fault-tolerance techniques used in traditional distributed architectures are often static, require special infrastructural support, and restrict the functionality of agent frameworks [14]. Instead, multi-agent systems should employ dynamic properties of their agents in order to provide efficient, domain-independent fault-tolerant solutions.

EXtensible Java-based Agent Framework (XJAF) [9] is our own FIPA-compliant [5] MAS. It consists of a set of loosely-coupled modules called *managers*. Each module is responsible for handling a distinct part of the overall agent management process. Managers can be added dynamically and are recognized solely by their interfaces, allowing custom implementations. Multiple distributed instances of XJAF can be connected to each other, forming a tree-like network structure, with the primary instance representing the root of the tree.

XJAF has a built-in support for agent mobility. While pursuing their goals, agents can move freely among the connected instances of the environment. The process of agent location tracking is achieved through *forwarding pointers* [18]: when an agent moves from one XJAF instance to another, the originating instance keeps a migration pointer to the target instance. An agent can then easily be found by following a path through the tree of instances.

When it came to fault-tolerance, the original implementation of XJAF had two major drawbacks. First of all, the tree-like structure of interconnected instances was very fragile, as the breakdown of any one node would divide the network into two sets of mutually unreachable systems. There was also no easy way of reintroducing the repaired MAS to the network. Secondly, as the forwarding pointers technique kept no track of alternative routes, a migrating agent would have become lost if any intermediary MAS in its path broke. The implementation of location tracking was also not robust enough, as it, for example, did not include cyclic path management.

Our main motivation behind the presented work was to alleviate these problems and, by doing so, to improve fault-tolerance of our system. We propose a solution based on the introduction of two types of specialized mobile agents. We have replaced the tree-like structure of interconnected XJAF

instances with a fully-connected graph, and one of the main tasks of the introduced agents is to allow efficient construction and maintenance of the new organization. Additionally, the newly introduced agents have the job of building a fault-tolerant location tracking system, one that will allow for robust agent tracking mechanism. And although our initial goal was to improve the fault-tolerance of XJAF, our solution based on light-weight mobile agents can easily be applied to any MAS, as it is framework-independent, requiring only few changes to the underlying MAS implementation.

The rest of the paper is organized as follows. In Section II we provide an overview of the related work. Section III describes our approach to building and maintaining a network of multi-agent systems in greater detail. Section IV presents our solution to the robust agent-tracking problem. Finally, in Section V we draw a conclusion of the presented work and propose some future enhancements.

II. Related work

Agents are often used as low-level tools for assuring conditions and providing support for higher-level processes. For example, *middle agents* [6], [17] enable efficient flow of information, by acting as yellow page servers, data integrity protectors, and intermediaries between information requesters and information providers. FUSION@ [3] employs deliberate agents that perform intelligent load balancing, manage all communication to and from the MAS, supervise the integrity of other agents, etc. The main advantage of these approaches is flexibility, as existing agents (that is, functionalities) can be modified, and new agents can be added, all without disturbing the remaining parts of the system.

The benefits of using mobile agents in network management have been presented in [1], [12], and [16]. It has been shown that mobile agents enable flexible, dynamic network building, provide for efficient discovery of network elements that violate normal behavior, and offer remote maintenance features. As opposed to centralized administration systems, distributed agent architectures that host and dispatch light-weight mobile agents increase network management efficiency by reducing the traffic, especially in unstable network environments.

Significant part of the research effort regarding MAS fault-tolerance is oriented towards increasing robustness and failure resistance of agents themselves. This problem can usually be handled with *replication* techniques, such as those described in [2], [13], and [19]. The simplified idea behind these techniques is to keep multiple copies of an agent, distributed across a number of frameworks. In case the original agent fails, one of the copies automatically takes over its task execution process. Although agent fault-tolerance is beyond the scope of this paper, we apply the basic concepts behind the replication techniques to achieve robust agent tracking.

Fault-tolerance of a multi-agent system based on middle agents can be improved with a *swarming controller* [4]. The purpose of the controller is to reduce the chance of an agent becoming isolated. For this task, it relies on two components: (1) a *population manager*, which maintains an optimum number and distribution of mobile middle agents, so that there is always at least one alternative path for two agents to communicate; and, (2) an *information propagator*, which handles knowledge distribution, assuring that if any agent does become isolated, it can still access enough information to continue uninterrupted execution.

DimaX [10], [19] is an agent framework with a advanced, built-in fault-tolerance techniques. It uses interdependence graphs for evaluating the significance of every agent in the system, and maintaining an optimum number of replicas accordingly (process known as *adaptive replication*). The system also employs *host monitors*, low-level agents that try to perform early detection of failures and inform all interested parties about the problem in a timely manner. This would, for example, allow an agent and its replicas to leave the troubling environment, or to move important data from it. At the basis of the monitoring is a *heartbeat* technique, which agents in the system use to indicate their valid operational status to each other.

When it comes to fault-tolerance, certain aspects of the DimaX system are superior to ours. However, DimaX was designed and built around the concept of fault-tolerance. Our goal was to instead propose light-weight solutions that could be easily integrated into any existing MAS.

The problem of failed intermediary nodes in a forwarding pointers-based agent tracking system was addressed in [8]. The proposed directory service requires that every agent remembers up to n previously visited hosts, while each framework keeps track of up to n different positions of the agent. As the agent moves, these data sets are used to propagate information about the new host, providing a way of building alternative paths to the agent.

This solution to the problem of forwarding pointers is similar to ours. However, our solution offers increased flexibility. First of all, we assign handling of all transition-related information to a new type of agent, releasing the frameworks and actual mobile agents from keeping track of the migration process. Secondly, in the solution proposed in [8], n has to be chosen at design time, usually by analyzing the environment in which the system executes. We argue that it is difficult to choose a good value for n, mostly because component failure rate is not constant over time. Instead, we adapt the number of alternative paths to the agent's behavior – the more an agent moves across nodes, the more mobility is important to it, and we increase the number of alternative paths to it accordingly. However, for unstable environments, we also allow for the increment rate of number of paths to be risen and changed at run-time.

The Adaptive Agent Architecture (AAA) [14], [15] is a multi-agent system in which middle agents called *brokers* forward requests to and responses from registered agents. Since an operational broker is crucial for the functioning of the entire system, AAA comprises multiple brokers forming a team. Behavior of all team members is guided by a *teamwork theory*, which drives them to perform tasks that are beneficiary to the entire team. For example, if a registered agent becomes disconnected because its broker had failed,

all remaining team members will have a joint commitment of reconnecting to the agent.

We have based the behavior of agents in our solution on the teamwork theory proposed in AAA – an agent that detects a failure of a MAS will have a commitment of informing other agents of the failure. However, in AAA, all brokers are located within the same system, and a broker can easily instantiate and add another broker. Our agents solve a more complex problem of creating and maintaining a network of physically distributed frameworks. In addition, we have included some optimizations of the teamwork theory. If an error occurs, not all team member will try to solve it. Instead, the order of problem solvers is organized by a simple procedure.

III. Building a reliable network

Distributed, interconnected multi-agent systems comprising mobile agents can provide efficient, flexible solutions for a range of problems, such as information retrieval, power supply management, telecommunication networks, etc.

XJAF has a built-in support for agent migration and a mechanism for distributed problem solving. Previously, multiple instances of the MAS were organized into a tree-like, hierarchical structure, as depicted in Fig 1. During start-up, each instance (except for the top-level, primary) would inspect the address of its parent node and then register itself with it. But, although easy to built, the tree structure was characterized by a single point of failure, that is a breakdown of any one node in the tree would make all of its sub-nodes unreachable by the rest of the system, and vice versa. In addition, the system did not allow for an easy reintroduction of a repaired MAS.

In order to overcome this problem, we have redesigned the inter-MAS connection system so that it organizes XJAF instances into a graph-like structure. The produced graph is fully connected. This approach is optimal as it introduces no significant overhead and it makes the process of finding a path to the target MAS a trivial issue.

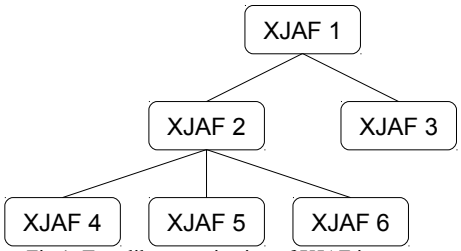

Fig 1: Tree-like organization of XJAF instances

Each MAS contains a list of all other systems in the network. A record in this list is a triplet (*Address, State, Timestamp*). *Address* is the physical address of the neighboring MAS, whereas *State* is its active state, as perceived by the current MAS (e.g. *running, unresponsive*, etc.). *Timestamp*s are an efficient way of selecting between mutually exclusive messages [8]. If an agent receives two messages about a MAS, one informing it that the MAS was started, other that it was shut down, the agent can use the timestamp of each message to determine whether the MAS is currently active (i.e. it was down, now it's up) or whether it is actually unreachable (i.e. it was up, now it's down).

The connection graph is maintained by a special type of mobile agent named *ConnectionAgent*. Each MAS in the network has a single instance of this agent residing in it. *ConnectionAgent* performs the tasks of adding a new MAS to an existing graph, as well as detecting a broken MAS and informing the remaining instances of the failure. As discussed earlier, the benefits of using a mobile agent in network management are lower demands on network traffic and system resources, and improved, distributed control, which becomes especially important in unstable environments.

A. Registering a new MAS

In the previous implementation, a newly created instance of XJAF, *X1*, would have had a single parent MAS to register with. In that configuration, however, a problem might arise if the parent MAS is not available (e.g. due to a failure) during the start-up of *X1*. So instead of having a single parent to register with, *X1* will now scan a predefined range of addresses until it finds the first available MAS. In order to optimize the scan, an administrator can narrow the range as much as possible, but at the same time keeping a sufficient number of possibilities.

After finding an available MAS, *X1* will create an instance of a *ConnectionAgent*, which then takes over the registration process. *ConnectionAgent* is a light-weight mobile agent that follows a simple registration algorithm. It migrates from one node in the network to another, spreading information about the newly created system.

ConnectionAgent keeps a list of all multi-agent systems it has detected so far. Upon arriving to a node in the network, the agent synchronizes its own list with that of its current host. As a result of the synchronization process, the host receives information about the agent's originating MAS, while the agent receives information about all systems the host is connected to. During this process, timestamps are used to select between mutually exclusive messages related to the same MAS, as described earlier. If the synchronization has resulted in any changes (i.e. the agent has learned about some new nodes in the network), *ConnectionAgent* will update its own list and send those changes to the originating MAS.

The agent then chooses next unvisited MAS from its list, and moves there. If there are no more unvisited systems, the process of adding the new MAS to the network has completed successfully, and the agent returns home. There, it continues to execute steps related to detecting a broken MAS, as described in the next sub-section.

Pseudo-code for the *ConnectionAgent*'s registration algorithm is as follows:

```
when (arrived_to_MASx) →
  mark_MASx_as_visited;
  new_MAS_list = synchronize_with_MASx;
  if (not empty(new_MAS_list))
  {
    update_my_list(new_MAS_list);
```

```
    ok = send_new_MAS_list_home;
  }
  else
    ok = ping_home;
  if (not ok)
    terminate_and_reverse_reg_process;
  next = get_next_unvisited_MAS;
  if (exists(next))
    move_to(next);
  else
    move_to(home);
```

After the synchronization process, the agent tries to communicate with its originating host. It will try to send it the list of any newly discovered multi-agent systems, or simply "ping" it, if there are no new systems in the result of the synchronization. In any case, the return value of this communication is used to indicate whether the originating MAS is still up and running or whether it had broken in the meantime. If the agent detects a failure of its originating MAS, it will terminate and reverse the registration process, as described in the next sub-section.

Fig 2 (a) depicts a scenario in which a new XJAF instance, *X1*, needs to be added to an existing network comprised of three frameworks: *A*, *B*, and *C*.

When the *ConnectionAgent* is created, only *C* will be in its list. After migrating to *C* and performing the synchronization with it, the *ConnectionAgent* will receive information about the two remaining instances, *A* and *B*. At the same time, *X1* will become registered with *C*. The agent will also send information about newly discovered systems to its originating MAS. This situation is shown in Fig 2 (b). The agent will then repeat the process two more times, visiting *A* and *B* and synchronizing information with them. Finally, it will have no more unvisited instances, meaning that the network is fully built, as shown in Fig 2 (c).

B. Detecting a broken MAS

Upon the successful completion of the registration process, the agent will return to its originating host, where it will continue to execute steps related to maintaining information about the state of the network. However, two problematic situations might occur during the registration: the originating MAS might fail, or the agent might become lost, e.g. due to failure of its current host.

In order to overcome these issues, *ConnectionAgent* needs to keep a heartbeat connection with its originating host. That is, it will communicate with the host at regular intervals, in, among other things, trying to detect its failure. The failure is handled by reversing the registration process – going back through the list of previously visited network nodes and informing them of the broken MAS. Similarly, if the originating MAS doesn't receive any communication request from the agent within a certain time threshold, it will initialize and dispatch a new instance of *ConnectionAgent*, restarting the registration procedure.

In a built network of multi-agent systems, *ConnectionAgent*s maintain a heartbeat connection to each other, in order to detect failures. Being guided by the teamwork theory, an agent that discovers a problem will inform other agents about it.

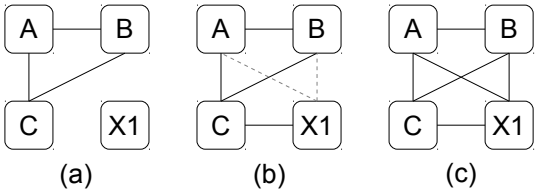

Fig 2: Adding multi-agent system X1 to an existing network

Agents execute asynchronously and independently, so many instances can detect the same failure at the same time and start informing each other, resulting in an unnecessarily increased usage of bandwidth and system resources. To prevent this from happening, we introduce an algorithm for controlling the heartbeat connection.

Multi-agent systems in the network are ordered by their respective timestamps (i.e. start-up times). We will say that *A* is *above B*, if it has earlier timestamp than *B*. Similarly, *A* is *below B* if it has later timestamp than *B*. Each *ConnectionAgent* keeps a heartbeat connection only with agents directly above and directly below it. Therefore, failure of a MAS will trigger a reaction in two agents. An agent that is below the broken MAS will forward this information only to agents below itself. At the same time, it will establish a new heartbeat connection with the *ConnectionAgent* residing in the first available MAS that is above the broken one. Similarly, an agent that is above the broken MAS forwards the information to all agents above itself, and establishes a new heartbeat connection with the *ConnectionAgent* residing in the first available MAS below the broken one. By following this simple set of rules, *ConnectionAgent*s can easily detect and handle even multiple failures of a series of nodes.

While informing other agents of a broken MAS, *ConnectionAgent* again relies on timestamps. If, for example, the broken MAS is rebooted before the agent has finished informing other participants of the problem, there will be two agents operating in the network: one telling that the MAS is down, and the other telling that the MAS is up. But, because the second agent is carrying a later timestamp, the correct state can easily be selected.

IV. Fault-tolerant agent tracking

The forwarding pointers approach is an efficient and easy way of tracking agent's whereabouts. As shown in Fig 3, if an agent moves from XJAF1 to XJAF2, and later to XJAF6, there will be a pointer from the first to the second, as well as from the second to the third host in the path. The pointers are used to forward any messages directed to the migrating agent.

However, if XJAF2 (or, in general, any node in the agent's path) breaks, the agent will be lost. As there are no backup paths, the originating framework is unable to determine the current host of the agent. The same holds if only the communication link between any two nodes in the path fails.

Combination of the newly integrated graph-like organization of XJAF instances and the concepts behind replication

techniques described earlier can be efficiently used to develop a robust agent tracking system. The algorithm is, again, implemented in a form of an agent, named *RemnantAgent*, making the solution suitable for any MAS.

Each mobile agent in the system has one or more instances of a light-weight *RemnantAgent* assigned to it, with a single instance residing in each MAS node in the path. A *RemnantAgent* can be considered to be a light replica of the agent. It responds to two events triggered by the agent's migration. The first event is signaled when the agent moves from one node to another. The event is dispatched to all but the agent's target MAS. The handler of this event can be described by the following pseudo-code:

```
when (agent_moved_from_X_to_Y) →
  agent_here = false;
  locations_after_this.push_back(Y);
  all_nodes_in_path.remember(X);
  all_nodes_in_path.remember(Y);
  // propagate migration backwards
  foreach (N in all_nodes_in_path)
    if (not N in locations_after_this)
      inform(N, agent_moved, X, Y);
```

RemnantAgent holds the two sets of information: (1) an unordered list of all nodes in the agent's path, and (2) a list of locations the agent has visited after leaving the *RemnantAgent*'s node. The second list is used for iterative reconstruction of alternative paths, while the first list is used to propagate elements of the second list. That is, whenever an agent moves from one node to another, this information is propagated backwards to all nodes in the path. Back nodes are identified by belonging to the list of all nodes in the path, but not to the list of future locations. All *RemnantAgent*s that handle this event will propagate the information to all back nodes, which might seem as unnecessary and traffic-heavy. However, this approach is required as any node, or a link between any two nodes, might break at any time.

The second event triggered by the migrating agent is sent only to the agent's target MAS, signifying the agent's arrival. The handler for this event is as follows:

```
when (agent_arrived_from_X) →
  agent_here = true;
  locations_after_this.clear();
  all_nodes_in_path.remember(X);
```

Consider a scenario in which there are 4 multi-agent systems, *A*, *B*, *C*, and *D*, comprising a network, and there is an agent (marked as ☺) originating in *A*. In the pursuit of its goal, the agent moves from *A* to *B*, as shown in Fig 4 (a). The migration triggers a creation of a *RemnantAgent* in *A*, which remembers that agent went to *B* (indicated by → *B*). Each host builds the unordered list of all frameworks interactively, by being part of the migration, or by propagating migration between other hosts. So after the first step, both *A* and *B* have the same list (*A*, *B*): *A* has built its list by sending the agent to *B*, while *B* has built its list by receiving the agent from *A*.

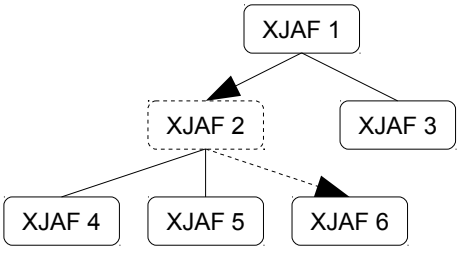

Fig 3: If XJAF2 fails, any agents that have through it are lost

Next, the agent moves from *B* to *C*, as shown in Fig 4 (b). *RemnantAgent* in *B* remembers the new position (→ *C*) and, after comparing this list with that of all nodes in the path (*A*, *B, C*), propagates the migration back to *A*, informing it of the change. *RemnantAgent* in *A* updates the position of the agent, keeping the previous value (→ *B*, *C*). If the agent continues to move, this time from *C* to *D*, the migration process will create another instance of *RemnantAgent* in *C* and trigger the location update in all previous instances. This situation is depicted in Fig 4 (c).

Now, suppose *A* needs to deliver a message to the migrating agent. *RemnantAgent* in *A* has the following list of locations: *B*, *C*, *D*. The last known location is *D* so it forwards the message directly there. But, let's say that there is a problem with the communication link between *A* and *D*, because of which the message cannot be delivered directly. To solve

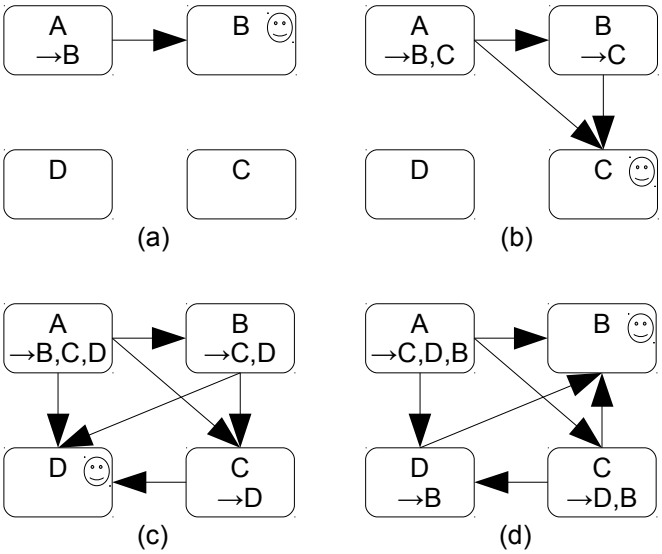

Fig 4: Robust agent tracking system

this issue, *RemnantAgent* in *A* inspects the previous host of the migrating agent and tries to deliver the message through it. So it forwards the message to *RemnantAgent* in *C*, which then, by looking at its own list of locations, delivers it to *D*. Similarly, if even *C* is down, the message can still be delivered, by going through *B*. In total, there are 4 alternative paths available:

1. $A \rightarrow D$
2. $A \rightarrow C \rightarrow D$
3. $A \rightarrow B \rightarrow D$
4. $A \rightarrow B \rightarrow C \rightarrow D$.

By going from the last known location backwards, each *RemnantAgent* will try to deliver the message in the shortest path possible.

Cyclic paths can also be easily handled. If, for example, the agent chooses to move from *D* back to *B*, the path tracking algorithm is similar, only this time *B*, as the current host, clears its list of locations, while all other nodes set *B* as the last known location of the agent. In addition, *A* will remove the previous instance of *B* from its list of locations. The final situation is depicted in Fig 4 (d). Again, there are 4 possible ways of reaching the agent from *A*:

1. $A \rightarrow B$
2. $A \rightarrow D \rightarrow B$
3. $A \rightarrow C \rightarrow B$
4. $A \rightarrow C \rightarrow D \rightarrow B$.

To improve the fault-tolerance of the tracking system even further, migration from each host can trigger the creation of *n* additional instances of *RemnantAgent*. These instances would then be distributed across different MAS nodes. Each new node can serve as an additional intermediary step in a path to the agent. The parameter *n* is configurable, and can be changed at run-time to reflect the overall system stability.

V. Conclusion and future work

Fault-tolerance of agent frameworks is an important issue. Large-scale agent-based systems need to be able to function in an uninterrupted fashion, assuring constant delivery of services to end-users. The underlying agent technology must, therefore, be capable of overcoming problems that emerge due to software or hardware failures.

The goal of the work presented in this paper was to offer a light-weight, yet efficient solution for improving fault-tolerance of existing agent systems. This has been achieved by the introduction of two types of mobile agents: (1) *ConnectionAgent* for building and maintaining reliable networks of distributed MAS instances, and (2) *RemnantAgent*, in charge of providing efficient agent tracking mechanisms by introducing robustness to the original forwarding pointers technique. All presented functionalities can be easily integrated in any existing MAS, with minimal changes to the underlying implementation.

Our future work on the subject of MAS fault-tolerance will be concentrated on providing flexible and robust agent replication and timely fault-detection techniques.

References

[1] A. Bieszczad, B. Pagurek, T. White, "Mobile agents for network management", In *IEEE Communications Surveys*, 1998.

[2] A. Fedoruk, R. Deter, "Improving fault-tolerance by replicating agents", In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pp. 737 – 744, 2002.

[3] D. I. Tapia, J. Bajo, J. M. Corchado, "Distributing functionalities in a SOA-based multi-agent architecture", In *7th international conference on Practical applications of agents and multi-agent systems*, PAAMS 2009, pp. 20 - 29, 2009.

[4] D. Šišlak, M. Pechouček, M. Rehak, J. Tožička, P. Benda, "Solving inaccessibility in multi-agent systems by mobile middle-agents", In *Multiagent and Grid Systems - An International Journal*, pp. 73 - 87, 2005.

[5] FIPA Homepage, http://www.fipa.org

[6] K. Decker, K. Sycara, M. Williamson, "Middle-agents for the Internet", In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.

[7] K. Jurasovic, M. Kusek, G. Jezic, "Multi-agent service deployment in telecommunication networks", In *Agent and multi-agent systems: technologies and applications*, LCNS Springer Berlin / Heidelberg, pp. 560 – 569, 2009.

[8] L. Moreau, "A fault-tolerant directory service for mobile agents based on forwarding pointers", In *Proceedings of the 2002 ACM symposium on Applied Computing*, pp. 93 – 100, 2002.

[9] M. Vidaković, B. Milosavljević, Z. Konjović, G. Sladić, "EXtensible Java EE-based agent framework and its application on distributed library catalogues", In *Computer science and information systems, ComSIS*, pp. 1 – 16, Vol. 6, No. 2, 2009.

[10] N. Faci, Z. Guessoum, O. Marin, "DimaX: a fault-tolerant multi-agent platform", In *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, pp. 13 – 20, 2006.

[11] R. Punithavathi, K. Duraiswamy, "A fault tolerant mobile agent information retrieval system", In *Journal of computer science*, Vol. 6, pp. 553 - 556, 2010.

[12] R. Stephan, P. Ray, N. Paramesh, "Network management platform based on mobile agents", In *International journal of network management*, Vol. 14, No. 1, pp. 59 - 73, 2004.

[13] S. Geraci, L. Giacalone, C. Leone, S. Mangano, G. Pitarresi, A. Scaglione, S. Sorce, A. Genco, "Fault tolerance", In *Mobile agents: principles of operation and applications*, edited by A. Genco, pp. 139 – 179, WIT Press, USA, 2008.

[14] S. Kumar, P. R. Cohen, "Towards a fault-tolerant multi-agent system architecture", In *Proceedings of the fourth international conference on Autonomous agents*, pp. 459 – 466, 2000.

[15] S. Kumar, P. R. Cohen, H. J. Levesque, "The adaptive agent architecture: achieving fault-tolerance using persistent broker teams", *Technical report: CSE-99-016-CHCC*, 1999.

[16] T. C. Du, E. Y. Li, A.-P. Chang, "Mobile agents in distributed network management", In *Communications of the ACM*, Vol. 46, No. 7, pp. 127 - 132, 2003.

[17] T. R. Payne, M. Paolucci, R. Singh, K. Sycara, "Facilitating message exchange through middle agents", In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pp. 561 – 562, 2002.

[18] T. Y. Yeg, T. I. Wang, "A ratio-based update scheme for mobile agent location management", In *Agent and multi-agent systems: technologies and applications*, Springer-Verlag Berlin / Heidelberg, pp. 100 – 109, 2009.

[19] Z. Guessoum, N. Faci, J. P. Briot, "Adaptive replication of large-scale multi-agent systems: towards a fault-tolerant multi-agent platform", In *Proceedings of the fourth international workshop on Software engineering for large-scale multi-agent systems*, pp. 1 – 6, 2005.

[20] Z. Yang, C. Ma, J. Q. Feng, Q. H. Wu, S. Mann, J. Fitch, "A multi-agent framework for power system automation", In *International journal of innovations in energy systems and power*, Vol. 1, No. 1, 2006.

[21] Z. Zhang, J. D. McCalley, V. Vishwanathan, V. Honavar, "Multiagent system solutions for distributed computing, communications, and data integration needs in the power industry", In *Proceedings of the General Meeting of the IEEE Power Engineering Society*, IEEE Press, pp. 45 - 49, 2004.