Communication Method for Reliable Multiagent Systems

Piotr Roliński
Insitute of Computer Science, Warsaw University of Technology

Abstract

The paper focuses on the communication inside a redundant multiagent system. We assume existence of one or several mirror agents for the selected principal ones. Our communication method supports an ordinary message exchange between agents and additionally, allows the communication with mirror agents. The communication process is explained in detail. We propose an extension for JADE (Java Agent DEvelopment Framework). JADE is an FIPA-compliant agent building platform for construction of interacting agents and multiagent systems. However, our extension can be applied also to any other agent development environment.

Keywords: multiagent systems, agent technology, agent communication, fault tolerance, error recovery

1. Introduction

The agent technology is a rapidly increasing area of research. The word "agent" has become very popular among artificial intelligence (AI) and computer science communities. The reason behind this popularity comes from the advantages that agent programming paradigm provide. Multiagent systems offer a way for construction of decentralised, emergent, concurrent and fault-tolerant architectures. An agent is an entity, which has attributes of intelligence, autonomy and perception. It perceives the environment it is situated in. Basing on observations, it is capable of selecting correct actions that will lead to accomplishment of its goals. The agent can react to changes in this environment and can adjust its own behaviour. Therefore, the agent should be naturally resistant or tolerant to failures. Nevertheless, the current research on multiagent systems concentrates on creation of agents, on their interactions with environment and with themselves. The fault tolerance aspect is disputed only in the terms what agents can provide, not how to make an agent tolerant to failures. This paper addresses the later problem.

The underlying concept of the method presented here is very simple and basic. Let's assume that a given multiagent system is dispatched over the unreliable environment, i.e., the one that can cause a failure of an agent. Each agent in this system has some concrete role and performs some tasks on behalf of its developer. The architecture or internal model of the agent(s) is not important to us - we are not interested how a given agent perceives, behaves and executes its actions (i.e., whether is it a Belief-Desire-Intention agent or purely reactive one or else). However, being a part of a group, the agent communicates with others in order to achieve some common goals. The communication takes place by a message exchange. Because of an unreliable environment any agent can fail and quit its execution (for example, when the host system on which a given agent is executed goes down). As a result, this agent stops sending messages and does not respond to any communication attempts. In such case, if the agent was involved in some global multiagent system task, this task cannot be completed.

Now, the idea is to make this particular system redundant i.e., develop multiple copies of each agent present in the system. These copies (mirrors) will be able to perform the same tasks as an original agent. They will be placed on the independent hosts to ensure the fault tolerance. The mirror agents can exist in the system from the beginning (from startup time) or be created in the moment of failure of the principal one. When the malfunction of a given agent would happen, other agents will start to cooperate and communicate with this agent's mirror. The redirection of communication and/or task execution may force the restart of the multiagent system (the calculations have to be repeated), or just may cause the rollback to some previous state of the system (when agents in the system support such an operation). In order to provide the described mechanism we have to do two things:

- offer the method for the detection of damaged/unresponding agents,
- give the means for the restart/rollback operation.

We assume that communication between agents happens either through peer-to-peer message exchange or through broadcasting. We do not discuss any special topology of connections – the assumption is that every agent can send a message to others. However, the communication between agents and mirrors is restricted – only the original agent can confer with its mirrors during normal execution of tasks. Of course, in case of failure of the original agent, other agents can (and should) contact its mirror. The last assumption considers the visibility of agents – we assume that every agent knows others ids (or names) and also mirror agents ids. Our method is implemented with use of JADE (Java Agent DEvelopment Framework) that is an FIPA-compliant agent building platform (see

[2], [3] and also [4]). However, the method is platform-independent and can be applied to any other agent development environment.

2. Concept

We assume that communication in the multiagent system happens through an asynchronous message exchange in an UDP-like fashion (i.e., via datagrams). The multiagent system consists of various principal agents. Each principal agent has several mirror agents (copies of the principal one) dispersed to different hosts. Two protocols are proposed here – one for replacement of a failed principal agent with its mirror, second for a mirror checkpoint update. First protocol takes into consideration the principal agents; the other refers to the single principal agent and its group of mirrors.

We propose to enhance the communication between principal agents with rather simple feature - *timeouts*. After the system startup, each principal agent starts counting down the *heartbeat time clock*. When the given time expires, agents broadcast *heartbeat messages* to other ones. These messages inform others that a given principal agent is still alive and working. After emission of such a message given agent resets its clock. All the agents gather heartbeat messages for a certain time period. Therefore, if some heartbeat message does not arrive in a timely fashion to a given agent, this agent assumes that the sender of the message is damaged. The timeout for receiving of heartbeat messages is specific to each agent (i.e., randomly chosen). This mechanism as well as the *restricted confirmation procedure* (explained later) allows to avoid the situation in which several agents detect a failure and all of them start a *replacement procedure*. In effect, only one agent, i.e., the one that detects the damage first, performs the replacement.

Nevertheless, the lack of the heartbeat message could mean either the sender has failed or the receiver agent is cut off from the communication lines. The second situation is easily detectable. If no heartbeat messages arrive to a given agent, this agent suspects it is itself out of order (perhaps its host was cut off from the network etc.). In this situation, the agent may rightfully expect that it will be replaced with the mirror agent, so it performs a *shutdown operation* (i.e., it ceases all its activities).

The lack of heartbeat message from a given agent detected only by one of its counterparts is not enough to state that this agent is damaged. In a hypothetical situation the heartbeat message can simply be lost in the network. Therefore, the agent, which did not received such a message, tries to confer about a supposed failure by broadcasting an "*ask about failure" message*. The broadcast starts the restricted confirmation procedure. The procedure is restricted, because only randomly selected, active (i.e., not damaged) agents answer the message, although, all agents listen to it. The agents that does not need to confirm the "ask about failure" message, simply stop gathering heartbeat messages and wait for the result of the confirmation. The chosen agents reply with a *"failure confirmed" message*. If they also detected the unresponding agent (i.e., they did not receive a heartbeat message from a disputed agent), they include damaged agent id in the message. Agents that do not perceive the suspected agent as damaged, send the same message, but without agent id. Such a message means a disagreement. The majority of messages of one type determine the agent reaction, whether to replace the agent in question or not. What more, the "ask about failure" message is broadcasted to all agents, including the supposedly damaged one, so this is the last chance for this agent to abort replacement operation by responding with a *"not damaged" message*. Note that the above procedures are not bounded to a detection of a single unresponding agent. Any number of damaged agents can be discovered in a single step (i.e., all the agents which heartbeat messages did not arrive) and their failure confirmed or not.

All of the messages sent inside the multiagent system are marked with time. Therefore, agents do not analyse messages older than a specified time/date. What more, agents do not wait for response messages infinitely. The *response timeout* specifies the time of wait for an acknowledgement. If the agent's heartbeat message does not arrive in time, this agent is assumed as damaged (the confirmation procedure starts). In case of lack of failure confirmation from the responder, its vote is simply omitted. When the agent receives no single response, it believes it has been separated from others and performs shutdown.

Time for timeouts should be chosen carefully, depending on the distance between agents in the system, as well as the communication traffic intensity. Inadequate timeouts may cause an erroneous situation. For example, the response message does not arrive on time (because of some lag on the network), but the agent itself is not damaged. As a result, the mirror agent starts to work in parallel with the principal agent. Both agents communicate with others, which can lead to disruption of the task execution. There are two solutions to such a problem. First, each agent depending on the state of the environment should recalculate the timeouts. However, we do not discuss such a mechanism in this paper. Second solution is to send a *shutdown message* to the suspected agent, in the first steps of the replacement procedure. When this agent receives such a message, it will stop execution of its tasks.

The *replacement procedure* consist of three steps:

- sending the shutdown message
- replacement of the damaged agent
- rollback to the last checkpoint

The agent that detected the failure is the performer of the replacement operation. The others participating in the confirmation procedure pause their execution just after sending the "failure confirmed" message and wait for the result of this operation. All other agents stop just after receiving the "ask about failure" message and also wait for an outcome. If the result of voting determines that a given agent was not damaged, the performer broadcasts a *"replacement cancelled" message* and all agents return to their activities. Otherwise, the performer sends the shutdown message to the damaged agent and executes the replacement task.

We have to consider two cases here: either a mirror agent (of the damaged one) exists from the beginning in the multiagent system, or it is created on the spot. In the first case, the replacement is simple: the performer sends the *activate message* to the mirror. The mirror becomes active and starts to accept all messages issued to the original agent. The second case requires a special construction of the whole system. The multiagent system (in the worst situation, every agent) has to hold some information about agent templates (like classes in object-oriented languages). The mirror is instantiated from such a template (object instantiation). In our implementation of the protocol, we take into consideration the first case.

As a result of the activation operation, the mirror agent is present in the system and sends back a "*activation confirmed" message* to the performer. The lack of such a message means that the mirror is also damaged. In such a situation the performer repeats the replacement procedure until an active mirror agent comes to live or the system runs out of mirror agents, which means general failure of the system. When all issued mirrors are active (or damaged) the performer broadcasts a *rollback message*. After receiving such a message, all agents including recently activated mirrors make a rollback, i.e., they return to a previously stored point of operation - they recreate their own states according to the states saved at the checkpoint. If agents do not support checkpointing, they restart their activities as at the startup time.

Note that if the mirror agent existed since beginning in the system and the system offers the checkpointing mechanism, this agent has to receive some updates about the latest checkpoint reached by its original agent. Thus, the principal agent, after the successful storing of its state, has also to bring up to date its mirror(s). Here, we offer the mirror checkpoint update protocol. The principal agent periodically (after saving information about checkpoint) issues a "*checkpoint update" message* containing the necessary information for an update. Mirrors acknowledge this message by issuing the *"update confirmed" message* after receiving and saving the update data. Lack of confirmation from the mirror agent indicates its failure. The original agent notifies everybody about this failure with a broadcast of a *"mirror failed" message* containing the mirror id. On the other hand, when there are no mirror agents present in the system since the startup time, there should be no checkpointing mechanism. In this paper, we do not discuss any method of checkpointing – we assume that the designer of the system will provide such a method. We only offer the protocol for propagation of checkpoint update data. In our implementation the rollback operation resets all agents to the startup state. However, we consider the enhancement of our method in future to include the checkpointing mechanism.

3. JADE

For an implementation and presentation of our method we have selected the JADE (Java Agent DEvelopment Framework) platform (see [3], [4]). This platform is a software development environment that offers tools for developing multiagent systems compliant to FIPA (Foundation for Intelligent Physical Agents) standards ([2]). JADE is fully implemented in Java language, which seems to be the best language for an object-oriented programming in distributed heterogeneous environments. The language provides the advantages of architecture neutrality and portability, so it is very suitable for construction of inherently distributed and sometimes mobile agents. JADE consists of a *Java packages* for development of agents as well as an *agent platform* (or runtime environment) for their execution.

The agent platform is fully distributed. It can be split among several hosts (provided they can be connected via RMI). Each host launches only one Java application, executing only one Java Virtual Machine. Agents run on hosts as Java threads and live within *Agent Containers* that provide the runtime support to the agent execution. The JADE supports the special *Remote Management Agent* that offers a graphical user interface for management and control of agents existing on a given host. The agent platform is FIPA-compliant, therefore it includes the *AMS (Agent Management Syste*m), the *DF (Directory Facilitato*r), and the *ACC (Agent Communication Channel*), which are activated at the startup phase. The Agent Management System is a special agent that provides a white page and life cycle service for registered agents, maintaining a directory of agent identifiers as well as agents' states. The Directory Facilitator is another agent, which offers a yellow page service, i.e., has capabilities to register, to deregister and to advertise the services of the DF-registered agent to the whole community. The developer can start many DFs composed into a federation in order to implement multidomain applications.

The Agent Communication Channel (Message Transport System) is a component that controls all the message exchange within the platform and between remote platforms. The messages themselves utilise the Agent Communication Language (ACL) – the language based on *speech act theory* [5], similar to KQML (Knowledge Query and Manipulation Language) [1]. The language uses the concept of performatives to allow an agent to convey its beliefs, desires and intentions. Therefore, a given ACL message consists of a sender id, receiver id, a performative specifying intent, a chosen content language (for example PROLOG) and an ontology containing semantics of the expression given in the content language. The semantics can be defined in a special FIPA–designed Semantics Language (SL). JADE supports an efficient method for transportation of ACL messages inside the same agent platform - messages are transferred encoded as Java objects. When crossing platform boundaries, the message is automatically converted to/from the FIPA compliant syntax, encoding, and transport protocol. The conversion is transparent to the agent developers - they only need to deal with Java objects. Agents exchange messages according to defined interaction protocols. JADE offers a library of such ready to be used protocols based on FIPA standards.

The agent itself is represented as a Java class. The *Agent* class offers some means for interaction with the agent platform as well as methods for implementation of agent's activities. The computational model of the agent's activities is multitask, where tasks (or behaviours) are executed concurrently. Each functionality/service provided by an agent is implemented as one or more behaviours composing a hierarchy. The agent platform schedules the agent behaviours in a non-preemptive fashion. JADE provides an FIPA-compliant naming service - at start-up agents obtain their GUID (Globally Unique Identifier) from the platform. The framework supports also the intra-platform agent mobility, including the migration of a state and a code of the agent.

The Java packages allow specifying a given agent and even whole multiagent systems from scratch by inheritance of available classes. JADE offers a multitude of packages for:

- development of agents and agent behaviours (*jade.core*, *jade.core.behaviours*),
- specification of communication language ontologies and semantics (*jade.lang*, *jade.lang.acl*, *jade.lang.sl*, *jade.onto*, *jade.onto.basic*),
- definition of the Agent Management entities – like AMS or DF (*jade.domain*),
- creation of GUI for observation of agents and agent activities (*jade.gui*),
- specification of interaction protocols, as well as Message Transport Protocols (*jade.proto*, *jade.mtp*),
- and finally offers some tools bundled in a special package (*jade.tools*) that supports the agent development process.

4. Implementation

The aim of this work was to develop a ready-to-use Java package that implements described protocols. The package is an extension to packages offered by JADE environment. In order to enhance the reliability of developed multiagent system, the designer need only to inherit the classes provided in the package while constructing agents. Our package (called *safe*) consists of three main subpackages: *safe.core*, *safe.gui* and *safe.util*. The *safe.util* subpackage contains a single *SafeAgentUtil* class, which collects several static methods used by other classes in the main package. The *safe.gui* subpackage has also a single class – *SafeGuiAgent*. This class is analogous to the *GuiAgent* class from JADE. It envelopes all events referring to changes in a given agent's GUI (see [4]), but instead of inheriting from *Agent* class, it is a child of *SafeAgent* one. The biggest subpackage, *safe.core*, is composed of two classes: *SafeAgent* and *SafeAgentStructure*, as well as three more subpackages: *safe.core.behaviours*, *safe.core.messages* and *safe.core.parameters*.

The *SafeAgent* class inherits all standard methods of the JADE *Agent* class. Therefore, it permits to add/remove behaviours of a given agent, as well as to send and receive messages. Additionally, it provides the means for a shutdown, rollback and checkpoint update operations. The shutdown of the agent refers to deregistration from the AMS and destruction of its thread (the standard JADE *doDelete()* method is called). Since we do not offer the checkpointing mechanism, the adequate methods are left as abstract – for specification by the developer in the inheriting class. For the rollback operation, we also have taken into consideration the possibility of checkpointing. The developer can either define appropriate methods or rely on the present mechanism in which all behaviours of the agent are reset, back to the initial state. The agent's behaviours are stored on a list-like data structure to allow a quick access (standard *Agent* class do not offer this feature). During the startup phase, the SafeAgent class creates and registers an additional (to user-defined) behaviour – *SafeAgentGeneralBehaviour*. This behaviour gathers all the tasks referring to protocols proposed for our method. For the message exchange defined by these protocols the agent utilises a special tree-like data structure. This structure reflects the knowledge a given agent has about other agents' identifiers. The structure is declared inside the *SafeAgentStructure* class.

The *SafeAgentStructure* class preserves the GUIDs of principal and mirror agents existing in the system. Identifiers of a single principal agent and its mirrors are put into the list (Java *Vector* class), with the first element being the principal agent id. Therefore, each such a group composes one list. All the lists are again put into a main list. Such an arrangement allows efficient updating of the information about available and active agents and removal of damaged agents' identifiers. The id of the unresponding agent is simply removed from the head of the appropriate list and automatically replaced by an adequate mirror agent's id (mirror becomes the principal agent). The class' methods support quick extraction of mirror agents' identifiers (for the mirror checkpoint update protocol) as well as all active agents' ones (for the message broadcast).

The *safe.core.parameters* subpackage contains the *SafeAgentParameters* class that is consisted of constants needed by *SafeAgent* and behaviour classes. These constants include the values of various timeouts defined in our protocols, as well as the initial knowledge base of an agent (i.e., identifier structure). The *safe.core.messages* subpackage has also a single class – *SafeAgentMessages*. This class gathers all the necessary factory methods for creation and reception of the messages described in our protocols. The creation methods produce the adequate ACL messages (JADE *ACLMessage* class). We do not create any special language or ontology for agents. We assume that every agent developed basing on *SafeAgent* class will understand the content of a message referring to our protocols. The reception methods offer message templates (JADE *MessageTemplate* class) that permit to distinguish the received message.

Finally, the *safe.core.behaviours* subpackage consists of behaviour classes that support the execution of our protocols. The *SafeAgentGeneralBehaviour* class is a cyclic Finite State Machine (FSM), which performs all of the operations by activating appropriate subbehaviours. When a given agent is a mirror agent, this main behaviour activates only the *SafeMirrorFunctionsProtocolBehaviour*. The subbehaviour listens to incoming messages and executes adequate tasks – the activation of the mirror agent, the update of checkpoint data (both tasks include sending confirmation messages), the rollback operation or the shutdown. In case that an agent is a principal agent, several subbehaviours are run consecutively in a loop. First, if appropriate time has passed, the main behaviour sends the heartbeat message. Next, it activates *SafeReplacementPerformerProtocolBehaviour*. This subbehaviour defines tasks for gathering of heartbeat messages, and performing the replacement and rollback operations. The operations start when a timeout for receiving heartbeat messages ends and the supposedly damaged agents are detected. Therefore, this subbehaviour specifies the performer part of the protocol. The responder part is defined inside the subbehaviour called *SafeReplacementResponderProtocolBehaviour*. This subbehaviour is responsible for receiving and responding to messages ("ask about failure", rollback and shutdown) by running adequate tasks. It is executed after the performer one, when there is no need to start the replacement yet. Periodically, depending on the checkpointing mechanism implemented by developer, the main behaviour may activate *SafeMirrorUpdatePerformerProtocolBehaviour* subbehaviour. This subbehaviour executes actions corresponding to the mirror checkpoint update protocol. All of described behaviours are FSMs and utilise additional subbehaviours in appropriate states of execution. Subbehaviours are responsible for sending and receiving of messages. All receiving actions are nonblocking, but have timeouts. Therefore, if a given message does not arrive during receive action, a given subbehaviour does not wait and passes the control further. However, when a given timeout expires and the message has not arrived yet, some appropriate actions are executed.

The main behaviour performs its tasks in parallel to other user-defined behaviours. Therefore, in order to enhance the designed multiagent system with reliability features offered by our method, the developer needs only to construct an agent basing on the *SafeAgent* class.

5. Future work

We have described the method to increase reliability of multiagent interaction. We have shown simple protocols for replacement of damaged agents with their mirrors and have developed the Java package implementing these protocols. This package is an extension to packages offered by JADE environment. In order to enhance the reliability of developed multiagent system, the designer need only to inherit the classes provided by our package while constructing agents. However, the presented concept is very basic and bounded by several assumptions. There is much to be done yet to create a full-scale fault tolerant framework. The future research can consider different topologies of agent interconnections. The groups of agents and their mirrors can be joined in TMR (Triple Module Redundancy) structures, i.e., mirror agents are active and also perform tasks, consulting results with original agent; the output is determined by voting. The mirror agents can detect a failure of the principal one and replace it within the group. We have offered only a skeleton mechanism to support the checkpointing operation. In future work we will try to extend this mechanism to provide the checkpointing independent of the created system. The last idea is to allow agents to recreate mirror agents, so the actual number of mirrors present for a single agent will be constant. This technique will permit to avoid the general failure of the system, when the original agent and its mirrors are all damaged. We can also push this concept further – if the template of a given agent exists in the system, an agent can repair a damaged one by regenerating it from this template – no mirror agent are needed.

Bibliography

[1]. T.Finin and G. Wiederhold. *An Overview of KQML: A Knowledge Query and Manipulation Language.* Dept. of Computer Science, Stanford University, 1993.

[2]. *FIPA 97 Specification, Version 2.0.*), http: //www.fipa.org, Foundation for Intelligent Physical Agents (FIPA), 1998.

[3]. *JADE Programmer's Guide*. http://sharon.cselt.it/projects/jade, CSELT S.p.A., 2000.

[4]. *JADE Administrator's Guide*. http://sharon.cselt.it/projects/jade, CSELT S.p.A., 2000.

[5]. J.R.Searle. *Speech Acts: An Essay in the Philosophy of Language.* Cambridge U. Press, 1970.