DOI: 10.2298/CSIS120108028P

Implementing an eXAT-based distributed monitoring system prototype

Gleb Peregud[1], Julian Zubek[1], Maria Ganzha[2,3], and Marcin Paprzycki[3,4]

[1] Warsaw University of Technology, Warsaw, Poland
[2] University of Gdansk, Gdańsk, Poland
[3] Systems Research Institute Polish Academy of Sciences
Warsaw, Poland
<firstname>.<lastname>@ibspan.waw.pl
[4] Warsaw Management Academy, Warsaw, Poland

Abstract. Monitoring resource utilization in distributed systems remains of importance. This is especially the case in LAN-based distributed systems (and, in particular, in global Grid systems), where individual nodes can be (may need to be) added to and/or removed from the system at "random" moments. The aim of this paper is to report initial results of the project that aims at using Erlang-based software agents as a robust and flexible resource monitoring infrastructure. The implemented prototype is capable not only of collecting performance data, but can also detect certain network problems. Furthermore, an assessment of the eXAT agent platform, based on experiences gathered during prototype implementation, is included.

Keywords: Grid computing, resource monitoring, Erlang, eXAT, intelligent agents

1. Introduction

In computing, *Grid* is a term that typically refers to a group of loosely coupled computers, working in a dynamically created arrangement to reach a common goal [37]. Grid technology is used both to solve computationally intensive scientific problems, and/or to deliver needed resources (e.g. computing cycles, software services, or data) in commercial applications. Such systems, by the definition, are heterogeneous and geographically dispersed. Here, two types of Grid systems can be distinguished. First, a *Global Grid*, somewhat similar to volunteer computing systems (e.g. the BOINC infrastructure [6]). Second, a *Local/Desktop Grid*, which can be characterized, among others, by existence of designated administrators (for the whole Grid installation, or for each of its parts). Unfortunately, Grid systems (as well as other LAN-based distributed systems) are prone to problems originating, for instance, from the network infrastructure, configuration, etc. Furthermore, one of the common problems concerning use of Grid infrastructures is load balancing. Hence, the need for software tools, which can help system administrators to monitor the state of the Grid (be it local or global) and efficiently manage its resources.

One of the interesting ideas, put forward by leading specialists in the fields of *Grid* and *agent* computing was to combine the strength of both approaches to deliver the computing fabric of the future (see [36], for more details). In other words, the idea was to use intelligence of software agents to provide the "brain" for the computational "muscle" of the Grid infrastructure. While there exists projects like the *Agents in Grid* [46,26,45,27,44], which attempt at directly realizing this vision, here, we focus our attention on application of software agents as "intelligent monitors" within the Grid (as well as in other LAN-based distributed systems, including Cloud infrastructures). In this context, note that a number of cases of agent-based monitoring systems have been described in the literature for other application areas. For instance, agents where used to monitor network traffic [53], an experimental environment in a laboratory [52], as well as power systems [51].

The aim of our project was two-fold. First, to develop foundations for a robust, fault tolerant, extensible, agent-based Grid / LAN / Cloud monitoring system, capable of working without need for manual configuration. Furthermore, the proposed system was to be capable of inferring knowledge from gathered data and acting upon it. Note that, while we focus on the Grid as the main use case, *all* results presented here are immediately applicable to the infrastructures within the Cloud environments, as well as to standard LAN infrastructures. Therefore, in what follows, the term Cloud (or LAN) could have been used in place of Grid (with proper caution applied, and with reflection on consequences of such interchange). Second, to assess robustness and flexibility of the eXAT agent framework [63,61,58] applied to the task at hand. Here, the potential advantages of an Erlang-based, FIPA compliant, agent framework are to be judged against their actual realization in the eXAT framework.

2. Related work

The proposed system is designed to support Grid / Cloud / LAN administrators in their routine activities. The two main use cases considered in our work are: (1) detection of "connectivity problems" (e.g. disappearance of a node or a link), and (2) monitoring (and reporting) performance metrics of individual nodes (or their groups). The latter use case can provide foundation for autonomous load re-balancing.

Task of resource monitoring has been solved by the monitoring software like Nagios [39] or Ganglia [49]. Both projects are quite mature, ready to use in complex, real-world situations. They were written without employing agent model, using traditional programming paradigms.

Nagios is an all-in-one system, which is able to monitor every key part of an IT infrastructure: system metrics, network protocols, applications, services, servers, etc. It is a general tool, which can be applied to monitoring Grid infrastructures as well. Within the Nagios there is a lot of space for customization through custom plugins. However, use of the Nagios system requires extensive manual configuration, which may be inconvenient in a geographically distributed large-scale Grid infrastructures. Furthermore, the fact that ownership of various fragments of the (global) Grid belongs to different entities, makes any “global configuration” task much more difficult.

Ganglia is a more specialized software for clusters and Grids (possibly consisting of smaller clusters). It is designed to achieve low per-node overhead, focuses on gathering performance metric of each machine, and on generating statistics for the whole cluster. Ganglia requires little configuration to start working and, like Nagios, supports custom plugins. However, it is not easy to extend its functionality beyond the assumed one (e.g. add inferencing knowledge based on collected data, and acting on it).

Furthermore, both these systems depend on the existence of a central server (or a group of servers), gathering information from remote processes working on every node. Note that, while in the monitoring system’s nomenclature, those remote processes are often called agents, they are just clients for a central server, and they lack typical properties of agents (for a classical definition of software agents and agent systems, see [40]). Design with a centralized server leads to potential problems. When the application server (or the machine on which it is running) fails, data will no longer be gathered. Similar situation occurs when, due to a network failure, some connections are broken. Another drawback of these approaches, is a need to reconfigure servers, in the case of adding new nodes to the Grid (or node removal).

Finally, let us recall that we would like to develop a system, which is able not only to provide information, which can be inferred from metrics gathered from the LAN, but also to act on it. While it would be possible to develop such system on top of either Nagios or Ganglia, it would immediately involve problems describe above. Furthermore, it would add another layer of software into already crowded Grid system software stack. Therefore, we have decided to build a system that will use capabilities of software agents, avoid the above mentioned problems of Nagios and Ganglia, and be capable of providing the additional needed functionality.

In the multi agent systems world, most of the projects connected with grid computing focus on goals similar to the *Agents in Grid* project. Monitoring of the physical network infrastructure is usually out of their scope. Nevertheless, some of them use approach similar to the proposed one.

AgentScape [23] is a distributed middleware that supports large-scale agent systems. It has features of an agent platform, as well as those of a distributed agent operating system. Among its features it provides decentralized resource discovery. However, since the focus of the AgentScape system is to develop agent middleware for large scale distributed systems, this project is much broader in scope. Furthermore, the last update of the AgentScape software is from April 2011 and it is unclear what is the progress of development of the AgentScape 2.

Similar, but less advanced, project was MAGDA: Mobile Agent Based Grid Architechture [20]. It was build on top of JADE agent platform and facilitated creating Grid applications based on mobile agents. Monitoring of agents and system resources as well as service discovery and load balancing were planned. It provided support for collective communication and use spanning trees for effective broadcast over the Grid. The same approach was later used in our project. Note that, the last published reference to the MAGDA system is from 2006 and thus we have to assume that it is no longer pursued.

An interesting approach to service discovery is represented by the ARMS [25] project. It provides an agent-based resource management system for Grid computing. The system consists of homogeneous agents managing local resources and advertising them through the Grid. A special agent has a global view of the system and simulates other agents' performance during runtime. It uses the PACE performance prediction tool-kit and, based on computed metrics, optimizes the behaviour of other agents. However, according to J. Cao, the development of both the PACE tool-kit and the ARMS project stopped in 2002.

3. Proposed approach—overview

Taking into account the above considerations, let us outline the main tenets of our proposed approach. First, we use software agents to develop a framework for intelligent monitoring of the state of a distributed system. We envision that a single agent will be placed at each node the system. Such autonomous agent will be capable of acting both as a "client" and as a "server." As a result, it will be capable not only of monitoring the state of the node, but also of inferring knowledge about the state of the system (or its fragments) and act on this knowledge. For instance, in the case of load imbalance, it will be capable of initiating procedures leading to the load re-balancing (see, also [28]). To achieve these goals, the proposed system will be designed in such a way that information about the state of the Grid (nodes of a distributed system) will be spread among the agents. Therefore, the information will remain available (at least to some extent) even after network link (or Grid node) failure. Furthermore, adding new nodes will not require any reconfiguration, because agents will be able to discover themselves, communicate and share the load of monitoring of whole system (Grid) evenly between them. Finally, this design of the system will alleviate the potential problem of a single point of failure. Let us present now two simple use cases that we have implemented in the initial system prototype, to illustrate its features and properties.

The simplest use case is: monitoring basic metrics in a basic LAN environment, with a star topology. Due to the zero-configuration feature, deployment of the system in a standard LAN should be very easy to complete. Right after the system is deployed, its administrator should be able to access his local agent (via a web interface) and start receiving information about the state of the nodes in the LAN (e.g. in form of plots).

A more complex use case is: continuous monitoring of a LAN with topology different than the simple star. System should be able to detect problems with network links and distinguish between network link failures, node failures, and failures of monitoring agents. This functionality requires that the system contains redundant network links, or alternative paths in the network. In most cases, no manual configuration should be necessary to make the monitoring system work.

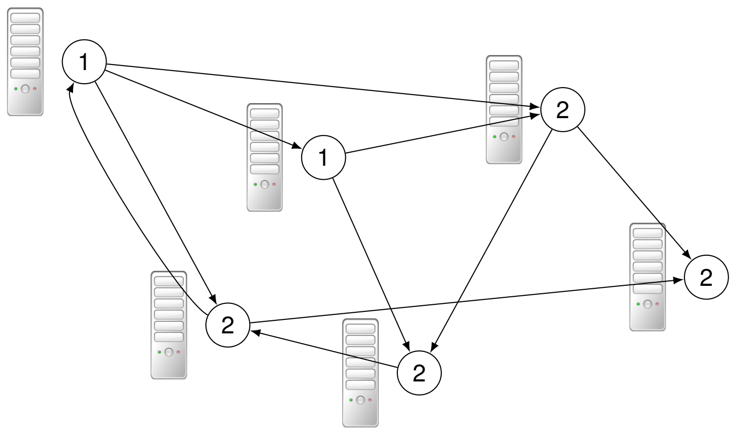


Fig. 1. Sample grid with monitoring agents. An edge from node A to B means that agent A monitors agent B. Number in circle is the number of agents monitoring agent running on the specific node.

To illustrate our approach, in Figure 1 we depict a sample network topology with agents running on every node. As we can see, there is no central node monitoring the remaining agents – the monitoring obligation is distributed, and every agent is monitored by at least one other agent. This illustrates how the robustness of monitoring process is achieved (this system will work correctly after the failure of a single node), and how the monitoring tasks are distributed evenly between agents, avoiding saturation of resources.

Recall that the proposed system is aimed primarily at supporting Grid managers (local administrators in the case of a local area network or a local Grid, as well as local and global administrators in the case of the global Grid). Therefore, as a starting point (to test the two use case scenarios), we have decided to implement the following features:

– access to the resource utilization metrics,
– automatic, zero-configuration discovery of agents in the local area network (or any other network where the multicast UDP is enabled), and
– inferring diagnostic information for basic problems, using a rule-based approach.

By the *resource utilization metrics* we understand various, easy to establish, metrics such as: CPU load, memory usage, running processes and their CPU usage, etc. Such data is going to be gathered and made available to the user. In the case of a human user, it will be presented to her as plots showing how the value of selected parameter changes over time. In the case of an agent user, data will be presented in an appropriate, machine-understandable format. For instance, in the case of the *AiG* project, such format is going to be derived from the ontology of Grid used there (see, for instance, [29,30]. This feature remains to be implemented.

For the two use cases, we decided to initially diagnose the following problems: (a) broken connections, (b) inactive nodes, and (c) inactive agents. Obviously, these are very basic problems, but they have been selected to illustrate capabilities of our approach to monitoring (e.g. they will show how a rule based expert system can be used within an agent to diagnose the nature of the network problem). Obviously, this list can be easily extended (e.g. by adding new rules to the expert system).

In our case, the *zero-configuration* means that no special information needs to be provided (by the user) for the agent to join the system. Therefore, when system administrator plugs her laptop into the network, her agent will find all other monitoring agents within it autonomously. As soon as this is done, the administrator agent can immediately start gathering information about the status of the Grid. Note that, the administrator agent, by default, runs the same code as all other agents with one exception—it is also running a GUI interface to display the obtained information in a human-readable format (plots via a web interface; see, section 4 for more details).

3.1. Agent technologies in the system

Agents and actors today Let us start from a brief methodological reflection. Observe that, recently the *actor model* of computing has (re)gained popularity. This due to the increasing complexity of building distributed software systems using more conventional models. Multiple "older" systems modelled after the actor model became popular in the industry, while new ones have been developed recently. Among them we can mention, Scala [50,24], SALSA [67], JavAct [17] and Kilim [60] in the Java world [42]; E language [56]; Asynchronous Agents Library [1] and Axum [5] from Microsoft; Act++ [41], Thal [43], libactor [8] and Theron [11] for the C/C++ languages; Stackless Python [65] and Stage [21] for Python; and Revactor [18] for Ruby. Some industry leaders have also employed the actor model, like usage of Scala by Twitter [34], and usage of Erlang by Facebook [47].

As we can see, the actor model is being adopted by a broad rage of companies, with at least some degree of success. Therefore, let us briefly discuss the relationship between actors and agents. According to Gul Agha, actors have the following properties [14]:

- concurrent computational entities
- independent from other actors (autonomous)
- can communicate with other actors via messages
- reacts to received messages

– can create new actors

According to Stan Franklin and Art Graesser, intelligent agents have the following properties [38]:

– reactive (or sensing; or acting)
– autonomous
– goal–oriented (or pro-active purposefully)
– temporary continuous (or continuous running processes)

Additionally they may display the following properties:

– communicative (or socially–able)
– learning (or adaptive)
– mobile
– flexible (actions not limited to simple script)
– character (referring to "personality" and emotional state)

As we can see, basic properties of agents match well with properties of actors. Hence we can argue that the *agents model* is a superset of the *actor model*, with addition of intelligence, goal-orientation, adaptiveness, mobility, proactiveness, etc.

Since Erlang, as a specific implementation of the actor model, has all above properties of actors build-in, we find it a good and natural foundation to develop an agent system. This is precisely what underlined the eXAT project [63] that we will now focus our attention on.

3.2. Erlang, eXAT and ERESYE

As stated above, we have decided to develop our system using the eXAT agent framework [63]. There were multiple reasons for this choice. First, according to its author (see [58]), the eXAT provides a FIPA-compliant implementation of an agent platform that includes:

– FIPA-ACL (Agent Communication Language),
– AMS (Agent Management System),
– support for ontologies,
– messaging using the MTP protocol,
– integration with the ERESYE (ERlang Expert SYstem Engine).

Since this looked quite interesting, we have decided to assess the quality of eXAT (which is an experimental tool) in practice of agent system development. Note also that, according to our vision of agent system design and implementation, Erlang, as an actor-based concurrent language with a distributed virtual machine, is really promising for implementing agent-based systems. Furthermore, being based on Erlang, could provide eXAT with certain advantages over other agent frameworks. They include: (i) natural, functional language syntax with declarative elements, suitable for representing knowledge; (ii) efficient agent communication and concurrency; (iii) easy access to Erlang libraries; and (iv) availability of a rule-based expert system engine build-in into eXAT.

One of often invoked characteristics of software agents is their intelligence. In our system this can be facilitated through the use of a rule-based expert system. We are referring to the ERESYE [62], which is a rule production system, written in Erlang and created by the same team that has developed the eXAT environment [62]. It is similar to other expert systems, like CLIPS [57] or Jess [13]. However, due to Erlang's declarative nature, it is possible to represent rules for the ERESYE using syntax very similar to the Erlang code itself. This simplifies the learning curve and allows use of the same structures for both communication between agents, and the reasoning subsystem. As stated in section 3, the ERESYE is to be used to infer information and provide it to the users of the system (e.g. system administrators).

Finally, note that, since the eXAT is claimed to be FIPA-compliant, and uses the FIPA-ACL message format, it should be possible to establish communication between agents written in eXAT and agents written in other FIPA-compliant systems, e.g. JADE [22]. This, in turn, should allow one to write systems, which utilize both agent platforms. For instance, it should be possible to add an eXAT-based monitoring subsystems to agent teams formed in the above-mentioned *Agents in Grid* project. Therefore, one of the auxiliary aims of our work was to establish that passing messages (bidirectionally) between eXAT and JADE is possible, without extra development efforts. Note also, that while implementing the system prototype (and the eXAT-JADE communication), as an extra result, we managed to made some observations comparing the Erlang/eXAT and the Java/JADE agent platforms, which we report in section 7.

4. Implementation details

Thus far we have implemented, a somewhat limited in scope, prototype of the above outlined system. This proof-of-concept implementation works in a LAN, which can be considered a basic variation of a Grid environment. In the near future, the system will be extended to support more complex environments. Let us now look into some details of the implemented prototype; starting from individual monitoring agents.

There are two basic functions of each agent in the system. First, monitoring other agent(s), and second, collecting local performance metrics. The third, extra function, is running a GUI, but it is executed only by those agents that are used to display data to the system administrators (see, below).

In our solution, to implement the *monitoring other agents* function, we use a method similar to that found in [45], and apply periodic pinging with FIPA-ACL *QUERY-REF* messages. Let us consider what information is needed for the agent operation. In our case this is:

- agent's own name (to provide correct reply address in sent messages),
- system metrics collected for the node,

– list of known nodes in the LAN,
– list of other agents which are monitored by given agent.

For an agent to be able to perform its monitoring tasks, it needs to gather the required information. Since the system assumes the zero-configuration approach, this has to be done automatically, using mechanisms, which are available at hand. Since agents are started independently on respective nodes, there is no pre-existing information about other agents in the system. Therefore, when choosing the name of the agent, which will be used to identify it in the system, we need to ensure its uniqueness. To achieve this goal, the name is generated automatically from the host name of the LAN (Grid) node. Here, we modified the eXAT code to use the "<nodename>.<hostname>" pattern for defining it's platform name. The *nodename* is the name of the Erlang node specified with the "-name" or the "-sname" parameter of the Erlang VM, while the *hostname* is the FQDN (Fully Qualified Domain Name) hostname of the system, as detected by the Erlang VM. The Erlang VM (and it's helper process *epmd*) ensure that there are no conflicting node names running locally. This ensures uniqueness of the "nodename" component of the agent name. The assumption that the host name of the node is correctly configured to be unique in the LAN, is sufficient guarantee of uniqueness of the agent name. Hence, since every monitoring node runs a single monitoring agent, we construct agent's name as "monitor_agent@<platform-name>". Additionally, a start script of a monitoring node retrieves list of locally registered Erlang nodes, and automatically selects a locally non-conflicting nodename.

System metrics can be acquired by using a suitable system library. Depending on the operating system used by the individual nodes, and the environment of choice, libraries like *parfait* [2] for Java, *glibtop* [3] for Linux systems coded in C/C++, or *Performance Counters API* [1] for Windows systems can be used. For the system prototype, we decided to use the *os_mon* ([33]) library, which comes with the standard Erlang distribution, and thus is a natural choice. Obviously, any of the above-mentioned libraries could be used, and interfaced with the monitoring agent. Here, the natural meta-encapsulation of the local information, which is the core of agent system development, is the guiding principle of our design. Additionally the os_mon abstracts all cross-platform details and exposes a consistent API for all platforms supported by Erlang. Therefore, let us make it explicit that the proposed monitoring system is operating system agnostic and will run also in a heterogeneous environment (consisting of computers running different OS'es) as long as all of them can run the Erlang VM.

List of all LAN nodes is discovered by using a DNSSD-based mechanism (as described in section 4.1). The remaining two lists can be built using a diffusion-based algorithm (described in section 4.3). In the near future, for more complex setups (e.g. in a distributed Grid), the agent discovery mechanism will be provided. This has to be done since the current algorithm assumes that multicast UDP is enabled, which is rarely the case in a non-LAN environment. If the monitoring system is going to be integrated into the project like the *Agents in Grid*, it can reuse mechanisms of agents discovery used in it (e.g. the Grid middleware).

Users of the monitoring system are provided with the monitoring data, using a web-base interface. Specifically, the user GUI adds to an agent a built-in web server. In our implementation, we have selected the Misultin framework [9]. It consists of two parts—a static HTML page, and a JavaScript code, which governs all logic of the agent's web UI. The implemented GUI uses Websockets to receive information from the agent (in real-time). The Misultin provides a ready implementation of the server-side WebSocket protocol, which we take advantage of. The web server is integrated into an agent in a form of one (or more) Erlang process(es), which communicate with agent's process(es) using Erlang messaging. This allows for a clean separation between the agent's logic and the code, which is responsible for sending this information to the browser. Here, the JavaScript library Smoothie Charts [10] is used for plotting the near real-time system utilization metric data in the browser.

Two activities, based on communication with other agents, are used in monitoring a group of neighbours are: (i) periodically pinging, and (ii) broadcasting information about the state of each agent (e.g. performance metrics, node status, link status, etc.) throughout the agent-Grid. Here, by broadcasting we understand sending an information to all other agents in the LAN. To accomplish this, without over-saturating the network, we propagate messages over the edges of a spanning tree. Creation and use of the spanning tree are described in section 4.2.

Last of core activities of each monitoring agent is diagnosing problems with nodes, links and other monitoring agents (see, also, section 3). This is achieved by application of the ERESYE expert system, and described in detail in section 4.4.

4.1. Agent discovery

Let us now consider how agents can find the list of other agents existing at any given moment in the system. Perhaps the most obvious solution would be a central registry (e.g. the AMS provided service), as it is used by default by many agent platforms including the eXAT. However, the central registry would be a single point of failure (SPOF) of the system. As a result, failure of the AMS node would, for all practical purposes, lead to disintegration of the monitoring system itself. Besides, it would not go well with our policy of zero-configuration (joining agent would be forced to communicate with the AMS to start working). Thus we decided to proceed the way that P2P systems collect information about nodes in the system, and avoid the SPOF [15].

In our system every node runs its own eXAT instance acting as an autonomous agent platform and registering only local agents. Information about other, external agents is collected and stored explicitly by every agent. To pass a message to another agent, we need to know the Internet address and the port number of the destination platform, and the destination agent name.

In this way, an agent registers another agent, when it stores the following information: address, port number and agent name. Obviously, this means that the amount of locally stored information is of order of the number of nodes in the Grid, but this is a “fair price” for the zero-configuration and avoiding the SPOF. Note that, agents entering or leaving the Grid should be registered (or deregistered) by all other agents running in that Grid. Recall, that since we have adopted the naming schema described in section 4, full agent name already encapsulates the platform name and the hostname.

Since, as mentioned earlier, the initial system is designed to work in the LAN environments (e.g. private Clouds/Grids) we have chosen a well-tested approach known as the Zeroconf [12], which is based on the UDP multicast and provides a DNS-like discovery system (multicast DNS—mDNS). The two most popular implementations of the Zeroconf techniques are Bonjour [16] and Avahi [4].

Bonjour is an Apple Inc. implementation of zero-configuration networks, including address assignment, service discovery and name resolution. It implements the DNS Service Discovery (DNS-SD), among others.

Avahi is an open source free implementation of the Zeroconf, including the mDNS and the DNS Service Discovery. Avahi is currently a de facto standard implementation of the Zeroconf for Linux and *BSD operating systems. Avahi also implements a source code API compatibility layer for Bonjour. Therefore, we have decided to use the Bonjour API, since it is available on all operating systems, where either Bonjour or Avahi are available.

In our implementation, we use the *dnssd_erlang* library, which provides an Erlang interface for the Bonjour API [66]. Each agent, during its start procedure, registers itself in the local DNS-SD registry (as a provider of the monitoring service) and receives a list of other monitoring agents. At the same time, it spawns a process, which listens for newly registered agents and adds them to the list. Note that the Avahi/Bonjour DNS-SD service, running in the operating system, automatically broadcasts information about all registered services to all computers in the LAN. It also automatically removes from this registry processes, which have been terminated. In this way, agents that leave the system are automatically deregistered.

This approach provides a much more flexible way of handling discovery of agents in the LAN than the eXAT AMS. In fact the eXAT AMS can be, with relative ease, extended to support the DNS-SD based agents discovery in the LAN. However, in our case, agent discovery is implemented directly in agents, without use of the AMS.

Knowing how agents can find information about other agents that are available in the system at any given moment, let us now describe algorithms that use this information and provide the monitoring infrastructure. In particular, we will provide details of broadcasting through the spanning tree, building the spanning tree, and the neighbour selection.

4.2. Broadcasting through the spanning tree

We had to make sure that agents can effectively broadcast information in the Grid, while the monitoring system is as non-intrusive as possible. In the case of a system consisting of hundreds of nodes, if a naive broadcast (exchanging information between every pair of agents) is used, network will become unnecessarily loaded. On the other hand, we had to ensure that the system will, with high probability, survive the failure of a single agent, node or a connection. In such case, the monitoring system should be able to send messages between the remaining agents and report accurately where the problem occurred. Furthermore, it should be able to successfully deal with leaving nodes (agents). Therefore, since we decided to use a spanning tree as the network topology of the monitoring system, this case should be handled in exactly the same way as if that node (agent) had crashed. When developing the logical monitoring network topology, we have taken into account the following issues:

- network saturation with messages exchanged between agents,
- need to detect and handle failures of:
  - limited number of nodes (in limited time),
  - limited number of links between nodes (in limited time).

The proposed solution consists of two stages. First to build a spanning tree of agents/nodes and to use it to broadcast messages. This guarantees that every agent will receive information just once (since the tree is by definition acyclic). Note that the information will be propagated regardless of broken links between specific nodes, as long as the network graph is connected (since in every connected graph the spanning tree exists). Second, to reasonably increase the number of agents monitoring each-other. In other words, the agent monitoring process should go beyond the basic spanning tree structure.

To build the spanning tree we use the well-known token-based distributed depth-first search [48] algorithm. Its core is based on passing a token along the edges of the graph; where the token contains the following information:

- state (FORWARD or RETURN),
- sender,
- list of visited nodes.

Upon reception of a token an agent undertakes the following actions:

1. If the token is in FORWARD state:
   (a) remember sender as parent
   (b) add current node to visited list
   (c) start children registration
2. If there are no unvisited nodes among the neighbours:
   (a) return the token in RETURN state to parent
3. Otherwise:
   (a) send token in FORWARD state to first unvisited neighbour
   (b) register that neighbour as child

One of the visible problems of this approach, when applied to monitoring of a dynamic system, is the need to rebuild the spanning tree in the case of a node, or a link, failure, as well as in the case of a node entering[5] or leaving the Grid. We allow the rebuild process to be initiated by any agent in the system. Such agent will initiate the spanning tree re-build as soon as it discovers that it no longer can communicate with one of its child nodes in the spanning tree. To discover it swiftly, an agent always monitors its children in the tree (through periodic pinging). Since many agents can initiate the tree rebuilding process at almost the same time (note that processes resulting in the need to rebuild the spanning tree can happen in multiple locations; e.g. a node failure and a node leaving the Grid can occur concurrently in various locations within the Grid), mechanism of breaking ties is needed (so that only a single spanning tree will result).

To implement such mechanism, we have modified the basic algorithm. Here, observe that each agent has its own unique identifier and thus it is possible to compare these identifiers using lexicographical ordering of their names. Thus, along with the token, we send the identifier of an agent that initiated the spanning tree rebuild process. Now, if an agent receives another FORWARD token before the RETURN token (that it has forwarded earlier), it checks whether the ID of the agent initiating another tree-rebuild process precedes the ID of the agent that initiated the previous one. If this is so, the previous FORWARD token is forgotten, which will lead to the previous rebuild process to time out, and yield no result, while the new rebuild process will continue, eventually reaching the previous search initiator. Otherwise the received token is ignored. After the spanning tree is rebuild, a special FINISHED message is propagated through it. Only after the reception of that message agents are ready to accept any further spanning tree search request (FORWARD tokens), regardless of the initiator ID. Such modification guarantees that in the case where multiple nodes initiate search concurrently, only one will succeed.

Spanning tree (re)built using this algorithm will be used for broadcasting data through agent's network. Current approach does not take into consideration the physical structure of the network over which agent's are communicating. This means that pings sent through agents' spanning tree may be actually traversing the longest possible paths in physical network and imposing unnecessary stress over network devices and network links.

We believe that automatic adjustments of a spanning tree based on the ping times between agents in a cluster (which is a case of an online minimum weighted spanning tree problem [54,31]) can be implemented and we speculate that it can be moderately effective for detecting the actual physical structure of the network for small networks, if proper statistics of ping measurements are used to determine the weights of edges in the graph. While this is a research topic in its own right, and is out of scope of the current system prototype, we plan to experiment with this approach in the future.

[5] Currently new agents "appearing" in the Grid are initiating a full tree rebuild, but this can be optimized in the future by attaching such agent as a leaf of the tree

4.3. Neighbour selection

Obviously, the spanning tree guarantees only that any missing node (or agent) will be detected immediately, since every agent pings periodically his parent and all his children, as a part of a standard monitoring procedure. However, it would be impossible to discover all broken links while monitoring only the edges of the spanning tree. If there is no direct monitoring between given two nodes, a broken connection between them would remain undetected; see figure 2. There, broken link between A and C will be discovered instantly whereas lack of connection between A and B is left undetected with the depicted spanning tree. Therefore, we have decided to force agents to monitor multiple nodes.

Since we have no "external" / a'priori knowledge about the physical network topology, we do not know if given two nodes are connected directly or if they communicate through another node (which may or may not have a monitoring agent running on it, e.g. a network router). Obviously, to monitor existence of connections, it is necessary to monitor direct connections between nodes. However, without information about the actual network topology, we cannot decide, which other nodes should be monitored by an agent. Therefore, we have decided to employ a probabilistic approach: each agent monitors a set of random neighbours. Here, we are satisfied that, with certain probability, all physical connections are covered. Obviously, the question remains, how many nodes, not belonging to the spanning tree, should an agent monitor?

To be absolutely certain that any broken network link will be detected (in the worse case, when each pair of nodes has distinct physical connection) each agent should monitor all other agents in the Grid. In some cases this would be acceptable, however, for larger Grids and sparse network topologies, it would lead to unnecessary network load (similarly to the naive broadcast considered in section 4.2). As a solution, we propose a parametrized value k, which defines a trade-off between the level of robustness we want to achieve, and the level of network saturation, which is acceptable in the current environment. Here, $k = 0$ means that no additional monitoring besides that through the spanning tree edges takes place, while $k = n$, where n is the number of nodes in the Grid, corresponds to the situation when every agent is monitoring all other agents.

To build the actual monitoring dependencies, we proceed as follows. Based on the list of all Grid nodes (which is available at each node, see section 4), each agent starts monitoring k random nodes. Choosing the exact value of k depends on network topology, throughput of it's links and the desired robustness of the monitoring (in a sense of ability of the monitoring infrastructure to detect complex network failures, and time needed to detect them). Unfortunately, to the best of our knowledge, the value of k should be selected experimentally. Higher k ensures better robustness, but increases bandwidth utilization, resulting from the monitoring process, and vice versa. At first, some of this monitoring is bidirectional, which leads to an unbalanced number of connections at each node. To make it closer to the requested value k, we employ a simple diffusion-based approach. Whenever two agents ping each other, they exchange information about the number of "neighbours" each has and compare them. If it turns out

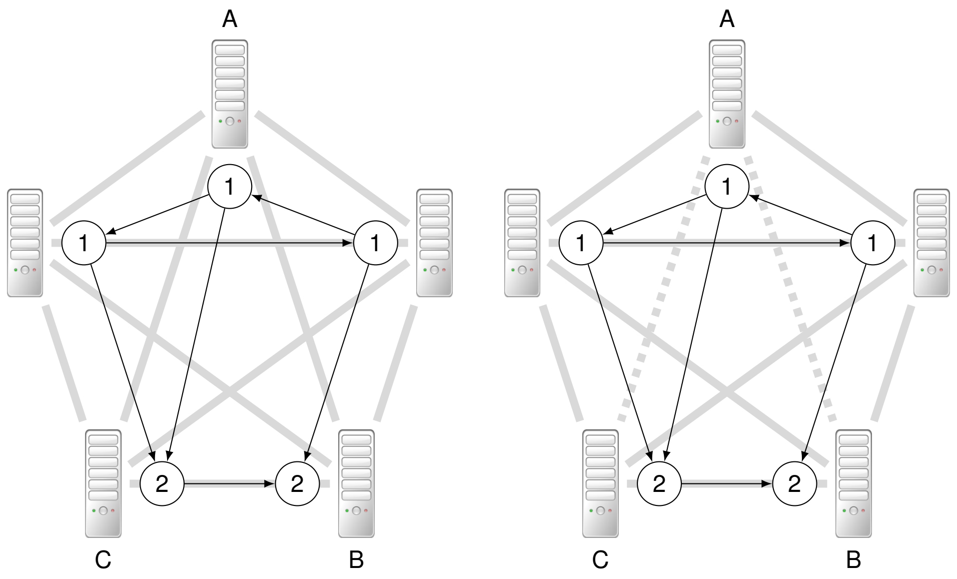


Fig. 2. Sample Grid with physical network connections. Thick gray edges denotes working network connections, dotted gray edges denote broken connections. Thin black edges denotes agents monitoring activity.

that one number is higher than the other (with difference larger than one), one agent will "pass" monitoring a specific node to the other. This will decrease the number of nodes it is monitoring and increase the number of nodes monitored by the other. To ensure data consistency, each exchange is realized as an elementary transaction, which has to be confirmed from both sides. After the diffusion is completed, in most cases monitoring ceases to be bi-directional and number of monitoring neighbours should be close(r) to the desired k. Our implementation of monitoring balancing is simplistic, and it may be slow to converge. Examples of better load-balancing algorithm of this type are well described in the literature (see, for instance [59]), and we plan to experiment with them in the future.

Furthermore, in the future versions of the system, this model will be slightly extended. When the number of neighbours monitored by each agent is stabilized, they will occasionally (after a predefined time) exchange monitoring obligations (pass one monitoring obligation and receive another). This will introduce neighbours rotation across the network. As a result, broken links that could not be discovered immediately, will be eventually discovered (after some time).

4.4. Knowledge inferencing

During work of the system, agents exchange information they have obtained (e.g. detected dead agents, detected dead nodes, performance metrics, etc.), using the broadcast through the spanning tree. Based on gathered cumulative information, agents are capable of interfering additional knowledge. Here, information gathered from other agents is added to agent's own knowledge base, which is used by it's expert system. These facts are also annotated with the name of the source agent. This allows to distinguish between information obtained first hand, and information obtained from other agents.

Let's consider three simple scenarios, which can be easily detected using a rule-based approach:

1. How to distinguish between a dead agent and a dead node?
2. How to distinguish between a dead node and a dead link?
3. How to detect problems of a specific feature of the system, which is running in the Grid?

For these scenarios, let us describe an appropriate set of rules, and their representation in the ERESYE inference engine.

- Case 1: We have an agent which monitors some remote agent *R* running on node *node(R)*, we can use the following rule to detect if the agent is malfunctioning, or it's the node that is dead.
  (agent *R* does not respond to pings) $\wedge$ (*node(R)* does respond to ICMP pings) $\Rightarrow$ (agent *R* is dead)

```
agent_failure(Engine, {agent_state, BNode_id, active},
                      {link_state, ANode_id, BNode_id, working},
                      {pinging_state, ANode_id, BNode_id, not_responding}) ->
  eresye:retract(Engine, {agent_state, BNode_id, active}),
  eresye:assert(Engine, {agent_state, BNode_id, inactive}).
```

(agent *R* does not respond to pings) ∧ (*node(R)* does **not** respond to ICMP pings) ⇒ (*node(R)* is unreachable)

```
node_unreachable(Engine,{link_state , ANode_id, BNode_id, broken},
                        {pinging_state , ANode_id, BNode_id, not_responding})->
   eresye:assert({node_unreachable, ANode_id, BNode_id}).
```

– Case 2: We have an agent, which monitors node *A* that stops responding to pings.
  (node *A* is unreachable)∧(node *A* is unreachable for other agents too) ⇒ (node *A* is dead)

```
node_inactive(Engine, {node_state, ANode_id, active})
   when not["{link_state ,ANode_id,_,working}"];true ->
   eresye:retract(Engine, {node_state, ANode_id, active}),
   eresye:assert(Engine, {node_state, ANode_id, inactive}).
```

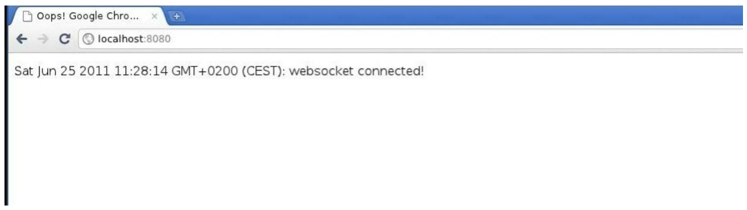
  (node *A* is unreachable)∧(node *A* is reachable for some other agents) ⇒ (link between *A* and myself is broken)

```
link_failure(Engine,{link_state , ANode_id, BNode_id, broken},
                    {link_state , ANode_id, _, active}) ->
   eresye:assert(Engine, {link_failure , ANode_id, BNode_id}).
```

– Case 3:
  For Case 3 let us assume that the system is installed in a heterogeneous Grid, where each monitored feature is handled by a subset of nodes. For example, let us assume that the Grid is handling back-end operations of some medium size web service, which has a search feature, which is being handled by 3 servers *A*, *B* and *C*. The fact that those specific hosts are handling some specific feature (in this case it is a search feature), can be fed to the knowledge base of agents in the monitoring infrastructure. Such information, would allow to create rules, which would detect that something is wrong with this specific feature. For example the following rule could be instantiated:
  (all nodes with this feature have high CPU utilization) ∧ (error rates for related category is elevated) ⇒ (the feature is broken)
  Coupled with an alerting system, this rule could generate an alert, informing administrators of the system that the search feature does not work correctly. As seen from the rule above, this example also needs an additional error rate metric originating from a monitoring system, which can be easily obtained by parsing error logs of said search servers. This can be implemented, for instance, by adding an additional agent, which will parse these logs and feed results to the local monitoring agent. Let us stress that to extend our existing prototype to handle this case would be relatively easy and would require only (a) adding appropriate rules to the expert system, and (b) adding (and integrating with other agents) a log parsing agent.

While these three cases are relatively simple, they were implemented to illustrate the eXAT ↔ ERESYE integration. Furthermore, these are the functionalities that not only can be helpful in actual day-to-day work of an administrator of services running on multiple servers in LAN, but also show that the implemented system works as assumed. Obviously, the set of possible rules is naturally extensible.

5. Monitoring system at work



Sat Jun 25 2011 11:28:14 GMT+0200 (CEST): websocket connected!

Fig. 3. Browser window after connecting to an agent.

Working with our system is very straightforward. We have prepared a wrapper script for running the agents:

```
./start_agent.sh -http_port 7778 -ws_port 8080 \
    -name platform_name
```

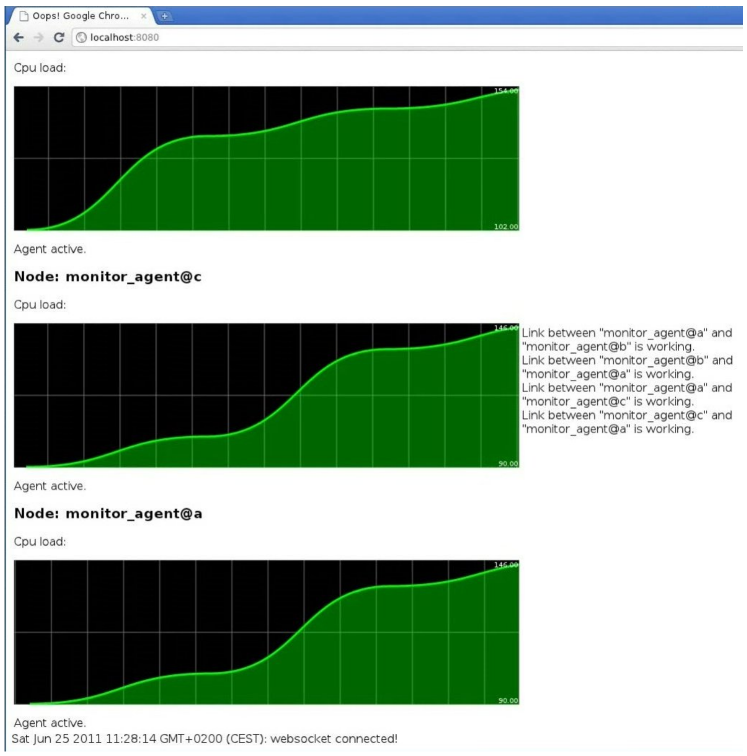
The http_port option denotes a port used by the eXAT for communication and the ws_port is the websocket port number used by the web interface. Finally, the $name$ parameter is mandatory for the platform name. However, it can be chosen freely, since it has no consequences for the operation of the system.

After the agent was started, the administrator can connect to it with a web browser and preview the collected information. Currently, only more recent versions of Internet Explorer, Firefox and Chrome support the WebSocket protocol, but since this support is likely to remain in these browsers in the future, we do not see it as a serious drawback. Figure 3 presents the browser window just after the connection with the administrator agent has been established.

Since the agent immediately starts gathering data, in a relatively short time user should be presented with animated plots presenting the CPU load of every node in the Grid (currently this is the metric, depiction of which has been implemented). This is illustrated in figure 4.

In the case of discovery of an inactive agent (as described in section 4.4), it is signaled by a red message appearing under its plot, as can be seen in figure 5.

Similarly, any link problems are signaled by a red text in the links state column. Since links are symmetrical, failure of a link from A to B would be always accompanied by failure of a link from B to A. This situation is depicted in figure 6.



Cpu load:

Agent active.

Node: monitor_agent@c

Cpu load:

Link between "monitor_agent@a" and "monitor_agent@b" is working.
Link between "monitor_agent@b" and "monitor_agent@a" is working.
Link between "monitor_agent@a" and "monitor_agent@c" is working.
Link between "monitor_agent@c" and "monitor_agent@a" is working.

Agent active.

Node: monitor_agent@a

Cpu load:

Agent active.
Sat Jun 25 2011 11:28:14 GMT+0200 (CEST): websocket connected!

Fig. 4. Resource load plots for different nodes.

6. Experimenting with the monitoring infrastructure

In addition to the simplistic use cases described in section 5, the monitoring system has been deployed in a network of 4 Ubuntu Linux workstations at a small company in Warsaw. After the Erlang has been installed (using apt−get install erlang), the monitoring system was unpacked and started with a provided startup shell script start.sh. Each node has been detected and was able to connect into the monitoring system in time below 5 seconds. The initial spanning tree is depicted in figure 7. To observe the monitoring system at work, we have implemented an additional monitoring mechanism, which recorded ping times along the edges of the spanning tree. These times were consistent with the physical layout of the network, which consisted of multiple 100 Mbps switches connected in a star topology.

After the system has been started it reported the CPU load of each computer in the network. Another metric, which has been presented to the user, was the recorded ping time between agents, which were pinging each other. Due to the mechanisms of broadcasting most locally harvested information, a user could look up the state of all nodes in the network at any node. Full knowledge base

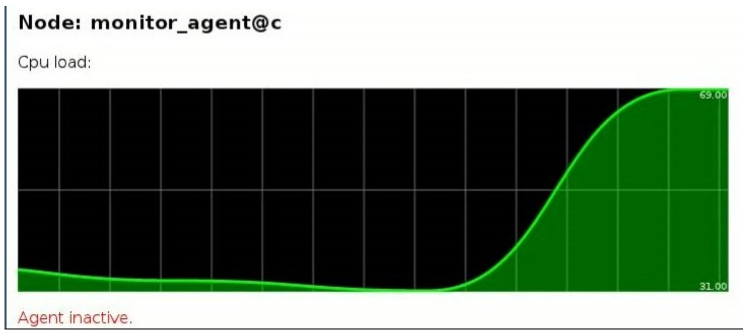


Fig. 5. Inactive agent discovered.

of one of the agents, including mentioned metrics, in the cluster is presented in figure 8.

When the root switch has been shut down, agents started to detect ping-timeouts, which resulted in creation a new spanning tree, as presented in figure 9. Agents were able complete this task in around 15 seconds, since each attempt of building a spanning tree resulted in multiple timeouts. A full knowledge base of one of the agents after shutdown of a switch is illustrated in figure 10.

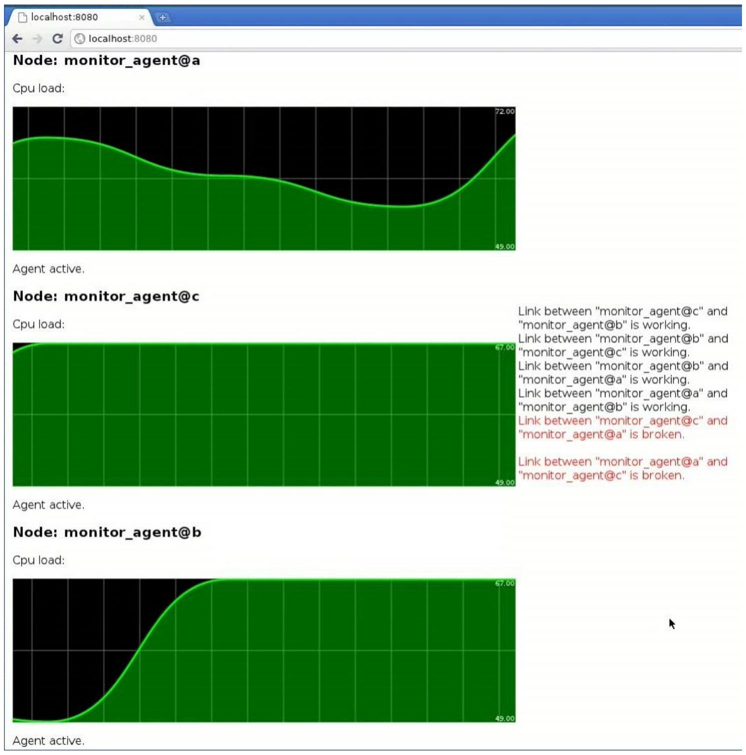
Separately we have collected measurements at the root switch, to estimate the bandwidth overhead introduced by our monitoring system. With the pinging interval set to 1 second, the average bandwidth overhead was approximately 70 kbps. Broadcasting overhead was changing linearly depending on the pinging interval.

After the root switch has been restarted, agents were able to detect their presence again, and were able to rebuild a new full spanning tree in around 7 seconds.

These experiments, performed in a realistic situation (though admitting, that the network was somewhat small) show that mechanisms outlined in the paper are working reasonably well and can be useful in practical use cases as a “smarter” alternative to software like Ganglia. Let us also note, that in the near future we plan to experiment with much larger network to test the scalability of the proposed approach.

7. Experiences with eXAT

Since no other projects realized in eXAT are known to us, and no descriptions of experiences regarding writing agents with this framework could be found in the literature, we have decided to present some thoughts on working with this tool.



Node: monitor_agent@a

Cpu load:

Agent active.

Node: monitor_agent@c

Cpu load:

Link between "monitor_agent@c" and "monitor_agent@b" is working.
Link between "monitor_agent@b" and "monitor_agent@c" is working.
Link between "monitor_agent@b" and "monitor_agent@a" is working.
Link between "monitor_agent@a" and "monitor_agent@b" is working.
Link between "monitor_agent@c" and "monitor_agent@a" is broken.

Link between "monitor_agent@a" and "monitor_agent@c" is broken.

Agent active.

Node: monitor_agent@b

Cpu load:

Agent active.

Fig. 6. Broken links discovered.

7.1. eXAT-JADE interoperability

Since eXAT uses the FIPA compliant ACL messaging model, it should be possible to communicate with other agents platforms supporting the standard. We verified this assumption experimentally, by establishing a bidirectional communication between an eXAT agent and a JADE agent. The connection proved to be easy to set and no platform modifications or special syntax were necessary (see, below). However, the original version of eXAT relied on a custom HTTP server which was not fully reliable. In our project we switched to an external solution — the Misultin [9], which helped with handling most of the JADE MTP messages. We haven't tested all possible communication types between the two systems, hence some other interoperability problems can still be present.

Despite possible inconveniences caused by the immaturity of eXAT, it seems to be feasible to build heterogeneous agents systems based on eXAT and JADE. Furthermore, using an eXAT agent to create an interface for the existing Erlang applications is also an option (which, for instance, could be used if eXAT agents would be incorporated into the *AiG* project).

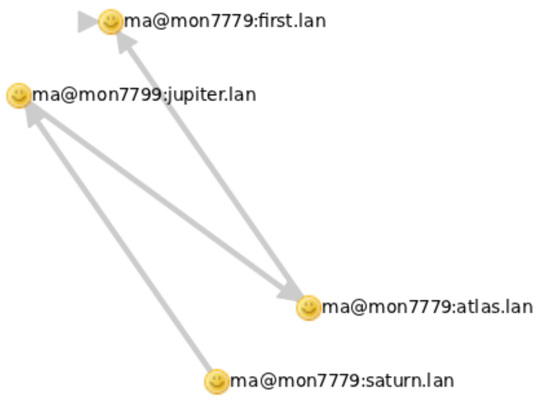


Fig. 7. Initial state of the spanning tree.

We shall now demonstrate how to make eXAT and JADE agents exchange messages. In the eXAT all ACL-related functions are defined in the module *acl*. The basic function for sending a message is acl:sendacl(message), where the *message* is the following Erlang record (defined in the acl.hrl header file):

```
#aclmessage {speechact, sender, receiver, 'reply-to', content,
language, encoding, ontology, protocol, 'conversation-id',
'reply-with', 'in-reply-to', 'reply-by'}
```

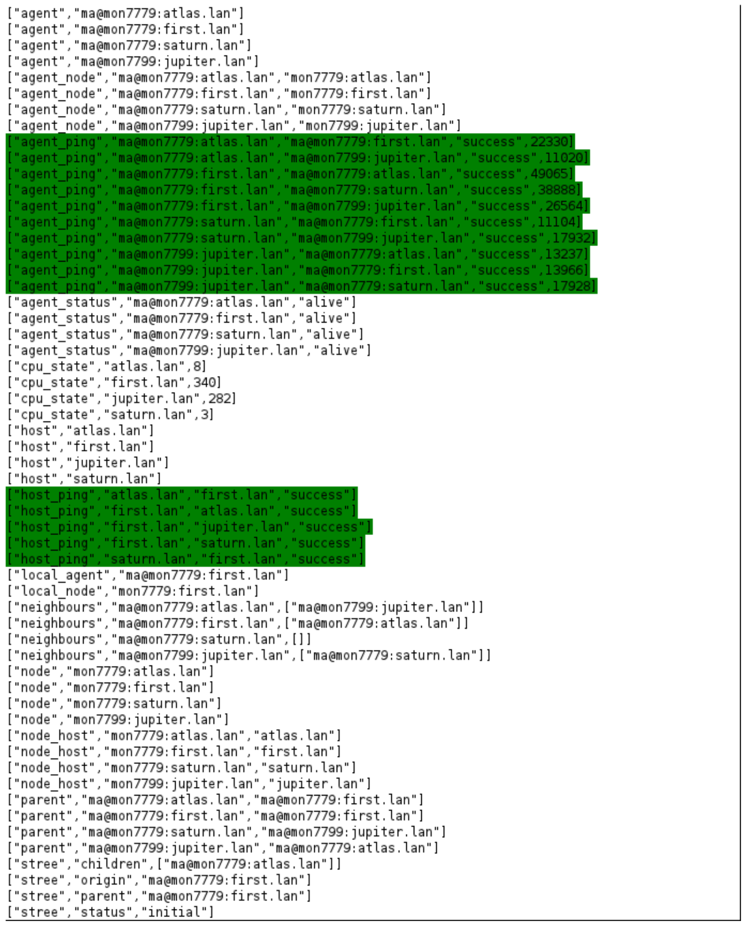
To pass a message between the agents we have to fill correctly the sender and the receiver fields. Both of them should be records of type:

```
#'agent-identifier' {name, addresses}
```

where the name has the form agent_name@platform_name, while the addresses is a list of network addresses in the form http://ip:port/acc. Finally, the #'agent-identifier' is defined in the fipa_ontology.hrl.

For example a command for sending simple ping message to an agent from a different platform (another instance of eXAT, or any other FIPA-compliant platform) running on the local host, could look as follows:

```
acl:sendacl(#aclmessage{speechact = query, content = "ping",
sender = #'agent-identifier'{john_the_agent@platform1,
[http://localhost:7778/acc]},
```



```
["agent","ma@mon7779:atlas.lan"]
["agent","ma@mon7779:first.lan"]
["agent","ma@mon7779:saturn.lan"]
["agent","ma@mon7799:jupiter.lan"]
["agent_node","ma@mon7779:atlas.lan","mon7779:atlas.lan"]
["agent_node","ma@mon7779:first.lan","mon7779:first.lan"]
["agent_node","ma@mon7779:saturn.lan","mon7779:saturn.lan"]
["agent_node","ma@mon7799:jupiter.lan","mon7799:jupiter.lan"]
["agent_ping","ma@mon7779:atlas.lan","ma@mon7779:first.lan","success",22330]
["agent_ping","ma@mon7779:atlas.lan","ma@mon7799:jupiter.lan","success",11020]
["agent_ping","ma@mon7779:first.lan","ma@mon7779:atlas.lan","success",49065]
["agent_ping","ma@mon7779:first.lan","ma@mon7779:saturn.lan","success",38888]
["agent_ping","ma@mon7779:first.lan","ma@mon7799:jupiter.lan","success",26564]
["agent_ping","ma@mon7779:saturn.lan","ma@mon7779:first.lan","success",11104]
["agent_ping","ma@mon7779:saturn.lan","ma@mon7799:jupiter.lan","success",17932]
["agent_ping","ma@mon7799:jupiter.lan","ma@mon7779:atlas.lan","success",13237]
["agent_ping","ma@mon7799:jupiter.lan","ma@mon7779:first.lan","success",13966]
["agent_ping","ma@mon7799:jupiter.lan","ma@mon7779:saturn.lan","success",17928]
["agent_status","ma@mon7779:atlas.lan","alive"]
["agent_status","ma@mon7779:first.lan","alive"]
["agent_status","ma@mon7779:saturn.lan","alive"]
["agent_status","ma@mon7799:jupiter.lan","alive"]
["cpu_state","atlas.lan",8]
["cpu_state","first.lan",340]
["cpu_state","jupiter.lan",282]
["cpu_state","saturn.lan",3]
["host","atlas.lan"]
["host","first.lan"]
["host","jupiter.lan"]
["host","saturn.lan"]
["host_ping","atlas.lan","first.lan","success"]
["host_ping","first.lan","atlas.lan","success"]
["host_ping","first.lan","jupiter.lan","success"]
["host_ping","first.lan","saturn.lan","success"]
["host_ping","saturn.lan","first.lan","success"]
["local_agent","ma@mon7779:first.lan"]
["local_node","mon7779:first.lan"]
["neighbours","ma@mon7779:atlas.lan",["ma@mon7799:jupiter.lan"]]
["neighbours","ma@mon7779:first.lan",["ma@mon7779:atlas.lan"]]
["neighbours","ma@mon7779:saturn.lan",[]]
["neighbours","ma@mon7799:jupiter.lan",["ma@mon7779:saturn.lan"]]
["node","mon7779:atlas.lan"]
["node","mon7779:first.lan"]
["node","mon7779:saturn.lan"]
["node","mon7799:jupiter.lan"]
["node_host","mon7779:atlas.lan","atlas.lan"]
["node_host","mon7779:first.lan","first.lan"]
["node_host","mon7779:saturn.lan","saturn.lan"]
["node_host","mon7799:jupiter.lan","jupiter.lan"]
["parent","ma@mon7779:atlas.lan","ma@mon7779:first.lan"]
["parent","ma@mon7779:first.lan","ma@mon7779:first.lan"]
["parent","ma@mon7779:saturn.lan","ma@mon7799:jupiter.lan"]
["parent","ma@mon7799:jupiter.lan","ma@mon7779:atlas.lan"]
["stree","children",["ma@mon7779:atlas.lan"]]
["stree","origin","ma@mon7779:first.lan"]
["stree","parent","ma@mon7779:first.lan"]
["stree","status","initial"]
```

Fig. 8. Initial state of agent's knowledge base.

```
receiver = #'agent-identifier'{tom_the_agent@platform2,
[http://localhost:7779/acc]}).
```

Here, an agent named *john_the_agent* sends a message to an agent *tom_the_agent*. These two agents are running on two separate platforms started on the localhost and distinguished by their port numbers.

Knowing the API of the *acl* module, we can create two simple agents – an eXAT agent (see listing 1.1) and a JADE agent (see listing 1.2)—and make them exchange messages.

Let us now analyse the listing in Figure 1.1. The eXAT code starts with the necessary header declarations. Next, follows the declaration of the function extends, which is a part of the eXAT object system syntax—it means that the

Listing 1.1. eXAT ping agent

```
-module(exat_agent).
-export([extends/0]).
-export([pattern/2,event/2,action/2,on_starting/1,
  do_request/4,start/0]).
-include_lib("exat/include/acl.hrl").
-include_lib("exat/include/fipa_ontology.hrl").

extends()-> nil.

pattern(Self, request)-> [#aclmessage{speechact='REQUEST'}].

event(Self, evt_request)-> {acl, request}.

action(Self, start)-> {evt_request, do_request}.

fellow_agent()-> #'agent-identifier'{
  name = "jadeagent@jadeplatform",
  addresses = ["http://localhost:7778/acc"]}.

on_starting(Self)->
  io:format("[Agent:~w]␣Starting\n", [object:agentof(Self)]),
  acl:sendacl(#aclmessage{speechact = 'REQUEST',
    content = "ping", sender = Self,
    receiver = fellow_agent()}).

do_request(Self, EventName, Message, ActionName)->
  io:format("[Agent:~w]␣Request␣received␣from␣agent␣~p\n",
    [object:agentof(Self), Message#aclmessage.sender]),
  object:do(Self, start).

start()->
  agent:new(the_exat_agent,[{behaviour, exat_agent}]).
```

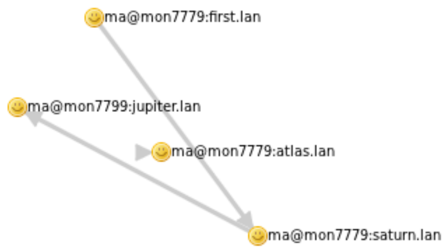


Fig. 9. Spanning tree after a switch shutdown.

agent class has no explicit parent. The next three definitions are necessary for the event mapping. The fellow_agent definition is a convenient way of storing the agent address—it is a functional equivalent of a global (class) variable. The on_starting is a function executed by the framework as soon as an agent is created. It contains the code for sending the message. The do_request is executed after receiving a REQUEST message. It returns a special construct, the object:do(Self, start), which informs that an agent is still in the state start.

Let us now take a look into the JADE agent code (in figure 1.2). Functionality of this agent is exactly the same as that of the eXAT agent, but the structure is slightly different. The setup is equivalent to the on_starting. There are no events for the message reception, so we have to manually fetch messages with the receive method. To do this in a loop-like manner, we exploit the JADE CyclicBehaviour.

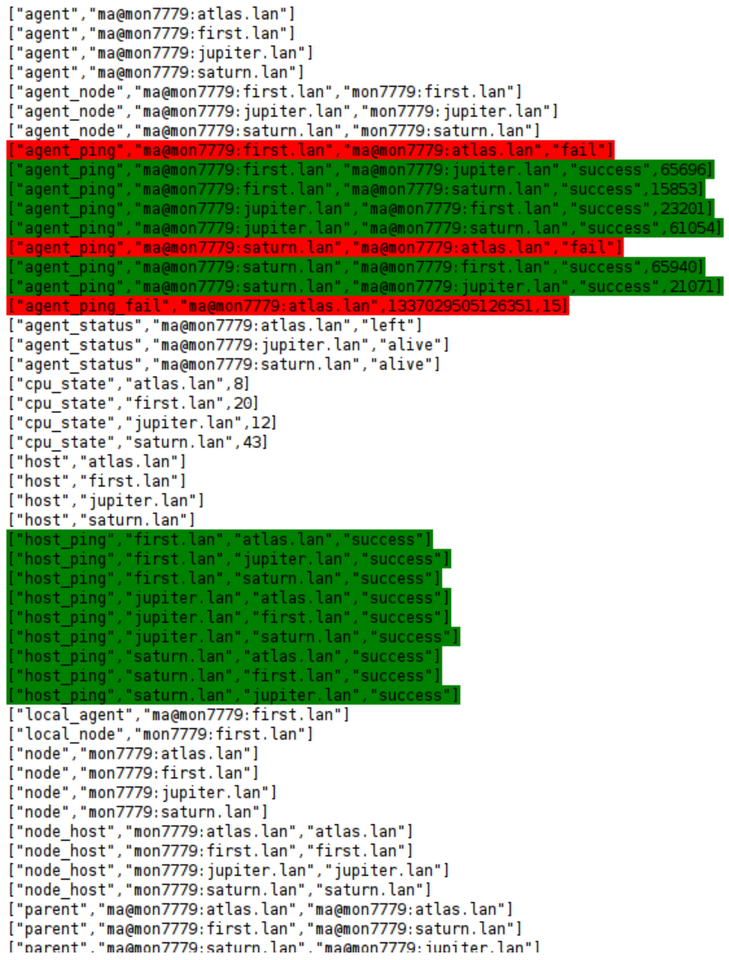
7.2. eXAT vs. JADE—implementation details

Let us now take a look at the eXAT and the JADE agent platforms side-by-side. Since JADE is substantially more mature than eXAT it, obviously, has a much richer set of features. Nevertheless, their basic FIPA-compliant functionalities are roughly equivalent.

One of JADE features nonexistent in eXAT is the possibility of sending Java objects in ACL messages, without special encoding, between agents of the

Listing 1.2. JADE ping agent

```
import jade.core.Agent;
import jade.core.AID;
import jade.lang.acl.ACLMessage;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.MessageTemplate;

public class JADEAgent extends Agent {

private MessageTemplate template =
   MessageTemplate.MatchPerformative(ACLMessage.REQUEST);

protected void setup() {
   System.out.println("[Agent:␣"+this.getLocalName() +
         "]␣Starting");

   addBehaviour(new CyclicBehaviour(this) {
      public void action() {
         ACLMessage msg = myAgent.receive(template);
         if (msg != null) {
            System.out.println("[Agent:␣" +
               myAgent.getLocalName() +
               "]␣Request␣received␣from␣agent␣" +
               msg.getSender().getName());
         }
         else {
            block();
         }
      }
   } );

   sendMessage();
}

private void sendMessage() {
   AID r = new AID ("the_exat_agent@exatplatform", AID.ISGUID);
   r.addAddresses("http://localhost:7779/acc");
   ACLMessage aclMessage = new ACLMessage(ACLMessage.REQUEST);
   aclMessage.addReceiver(r);
   aclMessage.setContent("ping");
   this.send(aclMessage);
}
}
```



```
["agent","ma@mon7779:atlas.lan"]
["agent","ma@mon7779:first.lan"]
["agent","ma@mon7779:jupiter.lan"]
["agent","ma@mon7779:saturn.lan"]
["agent_node","ma@mon7779:first.lan","mon7779:first.lan"]
["agent_node","ma@mon7779:jupiter.lan","mon7779:jupiter.lan"]
["agent_node","ma@mon7779:saturn.lan","mon7779:saturn.lan"]
["agent_ping","ma@mon7779:first.lan","ma@mon7779:atlas.lan","fail"]
["agent_ping","ma@mon7779:first.lan","ma@mon7779:jupiter.lan","success",65696]
["agent_ping","ma@mon7779:first.lan","ma@mon7779:saturn.lan","success",15853]
["agent_ping","ma@mon7779:jupiter.lan","ma@mon7779:first.lan","success",23201]
["agent_ping","ma@mon7779:jupiter.lan","ma@mon7779:saturn.lan","success",61054]
["agent_ping","ma@mon7779:saturn.lan","ma@mon7779:atlas.lan","fail"]
["agent_ping","ma@mon7779:saturn.lan","ma@mon7779:first.lan","success",65940]
["agent_ping","ma@mon7779:saturn.lan","ma@mon7779:jupiter.lan","success",21071]
["agent_ping_fail","ma@mon7779:atlas.lan",1337029505126351,15]
["agent_status","ma@mon7779:atlas.lan","left"]
["agent_status","ma@mon7779:jupiter.lan","alive"]
["agent_status","ma@mon7779:saturn.lan","alive"]
["cpu_state","atlas.lan",8]
["cpu_state","first.lan",20]
["cpu_state","jupiter.lan",12]
["cpu_state","saturn.lan",43]
["host","atlas.lan"]
["host","first.lan"]
["host","jupiter.lan"]
["host","saturn.lan"]
["host_ping","first.lan","atlas.lan","success"]
["host_ping","first.lan","jupiter.lan","success"]
["host_ping","first.lan","saturn.lan","success"]
["host_ping","jupiter.lan","atlas.lan","success"]
["host_ping","jupiter.lan","first.lan","success"]
["host_ping","jupiter.lan","saturn.lan","success"]
["host_ping","saturn.lan","atlas.lan","success"]
["host_ping","saturn.lan","first.lan","success"]
["host_ping","saturn.lan","jupiter.lan","success"]
["local_agent","ma@mon7779:first.lan"]
["local_node","mon7779:first.lan"]
["node","mon7779:atlas.lan"]
["node","mon7779:first.lan"]
["node","mon7779:jupiter.lan"]
["node","mon7779:saturn.lan"]
["node_host","mon7779:atlas.lan","atlas.lan"]
["node_host","mon7779:first.lan","first.lan"]
["node_host","mon7779:jupiter.lan","jupiter.lan"]
["node_host","mon7779:saturn.lan","saturn.lan"]
["parent","ma@mon7779:atlas.lan","ma@mon7779:atlas.lan"]
["parent","ma@mon7779:first.lan","ma@mon7779:saturn.lan"]
["parent","ma@mon7779:saturn.lan","ma@mon7779:jupiter.lan"]
```

Fig. 10. Knowledge base of one of agents after a switch shutdown.

same platform. This feature is not FIPA-compliant, but allows to greatly optimize communication between local agents. In fact one can use it to share persistent data across local platform. This is impossible to achieve with eXAT and only partially possible with standard Erlang messages (only one special type of binary data can be shared). While lack of shared memory is one of foundations of Erlang concurrency model (processes vs. threads), skipping unnecessary objects encoding is JADE's advantage over eXAT. Since this feature is only useful in local systems, we can assume that eXAT, like the whole Erlang, is designed to build systems distributed between physically different machines.

Keeping this in mind, and considering the threading model, we can speculate about ideal agent size encouraged by the frameworks. JADE authors suggest that there should be only a few “bigger agents” running within the platform, as each of them is connected with its own thread (in most Java implementations, a native thread) and increasing the number of agents would considerably increase the resource consumption. All the small-scale multi-tasking needed in the system should be based on JADE behaviours, which are lightweight and scheduled by the framework. However one has to remember, that behaviours of one agent have to be executed on one CPU as they belong to a single thread.

It is quite different in the eXAT, where all multi-tasking is based on the Erlang processes, running within the Erlang Virtual Machine (Erlang VM). They are lightweight and can be executed on any CPU (Erlang VM does the scheduling and workload balancing). This means that the eXAT agents can be as small or as big as needed, and this should not influence the performance of the system. Additionally the Erlang VM can handle millions of processes [68], hence there is no hard limits on the number of agents started in the eXAT.

7.3. Integration with ERESYE

Another topic that needs to be discussed is the ERESYE, and its integration with the eXAT. As stated above, the ERESYE is a full-feature rule-based inference engine implementing the RETE algorithm. It allows stating facts and specifying rules connecting these facts.

Facts are represented with standard Erlang data types, especially tuples. Rules are written using a normal function declaration form. The general syntax has been illustrated in section 4.4.

The inference algorithm is similar to an automatic logical inference, known from Prolog, but is optimized for the situation where asserted facts change dynamically, and exploits a form of eager evaluation. As soon as a new fact is asserted, all rules depending on it, are marked as partially satisfied. If the fact was the last not asserted prerequisite of a rule, the action is executed immediately.

It is also possible to synchronously wait for a certain fact to be asserted in the main code of the agent. This corresponds to an agent behaviour, which can be expressed as: “do not do a thing until you are sure that A is true.”

This approach makes possible effective reasoning, in real-time, in a dynamically changing environment. It is well-suited for the needs of responsive intelligent agents, as it is possible to use ERESYE as the central decision-making unit, which controls agent actions—a true equivalent of agent’s “brain.”

However, usage of ERESYE in eXAT goes further than that. Included tool makes it possible to translate an ontology (as defined by FIPA [55]) expressed in a simple hierarchical syntax, to a set of Erlang records suitable for storing as resolution engine facts. This is the basis for the so-called semantic layer of the eXAT platform. When the fipa_semantics_simple semantics is enabled for the eXAT agent, all “INFORM” speech acts are automatically added to the knowledge base of a given agent (called “mind” in eXAT). Additionally the fipa_semantics_simple is able to automatically check feasibility condition of an ACL message and perform a rational effect in accordance with the message’s communicative act. For example, a rational effect upon receiving a "CONFIRM" message is to assert it's content in the agent's knowledge base. With the semantic layer it is done automatically without any explicit message processing code [64]. For more information about handling of ontologies in ERESYE, please refer to [62].

In our opinion, the union between the eXAT and the ERESYE works very well. However, we found this system to be overly complicated to use. For instance, in our case, it was easier to explicitly use the ERESYE reasoner through it's API. However, we find the concept of integrating of eXAT agents with a reasoner, which is capable of parsing ontologies and processing ontological concepts, a very promising and interesting solution. Furthermore, integrating reasoners directly with the platform is something that should be considered also in other agent platforms.

7.4. eXAT not eXATly perfect

Unfortunately the eXAT, as an experimental tool, has some serious drawbacks. Some of them are our subjective opinions, stemming from thoughts on the eXAT design. This system is based on a custom implemented object-orientated emulation layer, which adds considerable complexity into the project. Constructing it feels a bit like "swimming against the current," because Erlang is designed as a purely functional language, and according to our beliefs, every agent aspect can be expressed in that fashion. Furthermore, it adds additional, unnecessary, overhead to message passing between (at least two) processes, which are used by a single agent. Finally, it adds an unnecessary learning curve, and decreases programming efficiency compared to the pure Erlang code, which is arguably a very efficient language to write in [69].

To clarify our opinion let us analyze a single case in depth. In eXAT, agents are modelled after the finite-state machine, which allows to bind different actions as a response to specific events, depending on the agent current state, thus introducing branching in code execution.

Listing 1.3. eXAT finite state machine description example

```
pattern(Self, request)-> [#aclmessage{speechact='REQUEST'}].

event(Self, evt_request)-> {acl, request}.

action(Self, start)-> {evt_request, do_request}.

action(Self, finalizing)-> {evt_request, decline_request}.
```

In 1.3, an agent executes the action do_request when in state start, but upon reception of the REQUEST message in state finalizing, it will respond with the decline_request.

For a finite-state machine, its single state is the only memory it has. In the case of a computer program, the state is distributed and represented as separate states of multiple variables. Each step of execution of a computer program can be a conditional clause, depending on the value of any stored variable. Therefore, the eXAT event mapping can be considered "syntactic sugar," i.e., element that does not bring new functionality, but makes the code easier to read. It is an alternative to wrapping event handling in explicit conditional instructions. However, Erlang has native mechanism for handling such cases, called pattern matching. It is possible to write multiple variants of the same function with different parameter patterns, and during the execution of the code, the correct version will be chosen, depending on the actual parameters. Therefore, the same goal is achieved, while maintaining brevity and sticking to Erlang's functional style.

Listing 1.4. Erlang finite state machine description example

```
handle_request(start, Request)->
  % do_request code
  return_value;
handle_request(finalizing, Request)->
  % decline_request code
  return_value.
```

The code in 1.4 handles or declines the request depending on the value of the first parameter. As we can see, the eXAT obscures this syntax with its own mechanism. We consider it a drawback, because it does not take full advantage of Erlang's strengths and provides a "replacement mechanism," which (particularly, for the Erlang programmers) may feel unfamiliar and unnecessarily complicated. To eliminate such complications introduced by the framework, we have implemented a simple_agent OTP behaviour, closely modelled after the OTP industry-standard gen_server behaviour[32]. It simplifies implementing agents, while following the Erlang coding style, and the OTP standards.

There are also other places where the eXAT was not written according to the Erlang/OTP coding standards, which are widely accepted in the Erlang community. Here are a few examples from eXAT source code demonstrating bad practices:

– Using non-OTP standard indentation style[7] and whitespace usage (spaces after function names):

```
inform␣(Message)␣->
␣␣sendacl␣(Message#aclmessage␣{speechact␣=␣'INFORM'}).
```

– Using lists instead of tuples as messages, and not using records for storing state of long-running processes:

```
handle_call([acl_erl_native, Acl], From,
            [AgentName, AclQueue, AgentDict, ProcessQueue]) ->
    %%io:format("[Agent] Received ACL=~w\n", [Acl]),
    {AclQueue1, ProcessQueue1} =
        perform_re(AgentName, AgentDict,
                   Acl, AclQueue, ProcessQueue),
    {reply, ok, [AgentName, AclQueue1, AgentDict, ProcessQueue1]}.
```

- Excessive use of *if* conditional, and using the *length* function where a simple *case* with pattern matching would be much more succinct and efficient:

```
handle_cast([getmessage, From],
            [AgentName,AclQueue,AgentDict,ProcessQueue])->
    if
        length(AclQueue) > 0 ->
            ProcessQueue1  = ProcessQueue,
            [Message | AclQueue1] = AclQueue,
            catch(From ! Message);
        true ->
            ProcessQueue1 = ProcessQueue ++ [{nil, From}],
            AclQueue1 = AclQueue
    end,
   {noreply,[AgentName,AclQueue1,AgentDict,ProcessQueue1]};
```

- Excessive defensive programming, discouraged by the Erlang Programming Rules and Conventions [35] and research [19]:

```
get_conds ({Module, Func}, Ontology, ClauseID) ->
  File = lists:concat ([Module, '.erl']),
  case epp:parse_file (File, ["."], []) of
    {error, OpenError} ->
     io:format(">> Errore!!!~n~w:~w~n",[{Module,Func},OpenError]),
      error;
    {ok, Form} ->
      Records = get_records (Form, []),
      %%io:format (">> Records ~p~n", [Records]),
      case search_fun (Form, Func, Records) of
        {error, Msg} ->
         io:format(">> Errore!!!~n~w:~s~n",[{Module,Func},Msg]),
          error;
        {ok, CL} ->
          ClauseList =
            if
              ClauseID > 0 -> [lists:nth (ClauseID, CL)];
              true -> CL
            end,
          %%io:format ("Clauses ~p~n", [ClauseList]),
          SolvedClauses =
            if
              Ontology == nil -> ClauseList;
              true -> eresye_ontology_resolver:resolve_ontology (ClauseList,
                                                                  Ontology)
            end,
          %%io:format (">>> ~p~n", [SolvedClauses]),
          case read_clause(SolvedClauses, [], Records) of
            {error, Msg2} ->
              io:format(">> Errore!!!~n~w:~s~n",[{Module,Func}, Msg2]),
              error;
            CondsList -> CondsList
          end
      end
  end.
```

- Occasionally implementing features, which are already present in the Erlang/OTP standard library. The following code duplicates the features of the ETS tables.

```
property_server (Dict) ->
  receive
    {From, get, AttributeName} ->
```

```
      case catch (dict:fetch (AttributeName, Dict)) of
        {'EXIT', _} -> From ! {ack, undef};
        Other -> From ! {ack, {value, Other}}
      end,
      property_server (Dict);
    {From, set, AttributeName, AttributeValue} ->
      From ! {ack, ok},
      property_server (dict:store (AttributeName, AttributeValue, Dict));
    {From, list} ->
      X = dict:fetch_keys (Dict),
      From ! {ack, X},
      property_server (Dict);
    {From, list_values} ->
      X = dict:to_list (Dict),
      From ! {ack, X},
      property_server (Dict);
    {From, exit } ->
      From ! {ack, ok};
    Other ->
      property_server (Dict)
  end.
```

Finally, note that the eXAT project was effectively discontinued by it's authors, since there were no updates to it since 2005; it has no community support, and no production systems using the eXAT platform could have been found. Furthermore, eXAT has a very sparse documentation. Additionally, it uses some custom libraries for well known tasks like implementing HTTP servers, when reusing existing libraries would lead to better tested and a more stable code.

While working on this project we made a number improvements to the eXAT, which can be found at the github.com/gleber/exat. Among others, we have replaced the internal custom-made HTTP server with the well-established Erlang implementation of the HTTP server called Misultin. As mentioned, we simplified creation of agents with the simple_agent behaviour. We switched to the rebar-managed compilation process, which is the current de-facto standard in the Erlang community. Finally, we have updated the eXAT to work with latest version of the Erlang/OTP distribution.

8. Concluding remarks

In this paper we have introduced an agent-based monitoring system for LAN / Grid / Cloud infrastructure. The prototype of the system has been implemented using an Erlang-based eXAT agent platform. After implementing the system, we found Erlang to be a good fit for the task at hand. Furthermore, the eXAT agent platform was acceptable as the tool to implement the monitoring system, though in needed at least some improvements (which we have completed). For the agent reasoning we have used the ERESYE rule-based expert system, natively integrated with the eXAT. This integration worked very-well and we plan to use it to cover more extensive cases of reasoning about the state of the system. In the near future we plan to: (a) fix additional issues with the eXAT, primarily continue refreshing it to match the state of the art of Erlang today (these improvements will be made available to the community); (b) expand the set of network topologies and detected problems; (c) integrate the eXAT monitoring system with the resource managing agents in the *AiG* project, and (d) complete additional research outlined in the paper. We will report the results of our work in subsequent publications.

References

1. http://msdn.microsoft.com/en-us/library/windows/desktop/aa373083
2. http://code.google.com/p/parfait/
3. http://developer.gnome.org/libgtop/stable/libgtop-GlibTop.html
4. Avahi homepage. http://avahi.org/
5. Axum - microsoft's actor programming language, http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx
6. Boinc project. http://boinc.berkeley.edu/
7. Erlang: submitting patches. https://github.com/erlang/otp/wiki/submitting-patches
8. libactor project documentation. http://www.chrismoos.com/
9. Misultin. https://github.com/ostinelli/misultin
10. Smoothie-Charts library. http://smoothiecharts.org/
11. Theron - c++ concurrency library. http://www.theron-library.com/
12. Zero Configuration Networking (Zeroconf). http://www.zeroconf.org/
13. Jess website. http://www.jessrules.com/ (2011), sandia National Laboratories
14. Agha, G., Hewitt, C.: Concurrent programming using actors: Exploiting large-scale parallelism. In: FSTTCS. pp. 19–41 (1985)
15. Aloisio, G., Cafaro, M., Fiore, S., Mirto, M., Vadacca, S.: Greic data gather service: a step towards p2p production grids. In: Proceedings of the 2007 ACM Symposium on Applied Computing (SAC). pp. 561–565. Seoul, Korea (2007)
16. Apple, I.: Bonjour overview. http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/NetServices/Introduction.html#//apple_ref/doc/uid/10000119i, december, 2011
17. Arcangeli, J., Maurel, C., Migeon, F.: An api for high-level software engineering of distributed and mobil applications. In: Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems. pp. 155–. IEEE Computer Society, Washington, DC, USA (2001), http://dl.acm.org/citation.cfm?id=874065.875773
18. Arcieri, T.: (2008), http://revactor.github.com/
19. Armstrong, J., Helen, T.: Making reliable distributed systems in the presence of software errors (2003)
20. Aversa, R., Di Martino, B., Mazzocca, N., Venticinque, S.: Magda: A mobile agent based grid architecture. Journal of Grid Computing 4, 395–412 (2006)
21. Ayres, J., Eisenbach, S.: Stage: Python with actors. In: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering. pp. 25–32. IWMSE '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/IWMSE.2009.5071380
22. Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.: Jade—a white paper. Tech. rep., Telecom Italia Lab, EXP Online (2003), december 2011

23. Brazier, F.M.T., Mobach, D.G.A., Overeinder, B.J., van Splunter, S., van Steen, M., Wijngaards, N.J.E.: Agentscape: Middleware, resource management, and services. In: Proceedings of the 3rd International SANE Conference (SANE 2002). pp. 403–404. Maastricht, The Netherlands (May 2002)
24. Bruno, Gibbons, J.: Scala for generic programmers. In: Proceedings of the ACM SIGPLAN workshop on Generic programming. pp. 25–36. WGP '08, ACM, New York, NY, USA (2008), `http://dx.doi.org/10.1145/1411318.1411323`
25. Cao, J., Jarvis, S.A., Saini, S., Kerbyson, D.J., Nudd, G.R.: Arms: An agent-based resource management system for grid computing. Sci. Program. 10(2), 135–148 (2002)
26. Dominiak, M., Ganzha, M., Gawinecki, M., Kuranowski, W., Paprzycki, M., Margenov, S., Lirkov, I.: Utilizing agent teams in grid resource brokering. International Transactions on Systems Science and Applications 3(4), 296–306 (2008)
27. Dominiak, M., Kuranowski, W., Gawinecki, M., Ganzha, M., Paprzycki, M.: Utilizing agent teams in grid resource management—preliminary considerations. In: Proc. of the IEEE J. V. Atanasoff Conference. pp. 46–51. IEEE CS Press, Los Alamitos, CA (2006)
28. Drozdowicz, M., Ganzha, M., Kuranowski, W., Paprzycki, M., Alshabani, I., Olejnik, R., Taifour, M., Senobari, M., Lirkov, I.: Software agents in adaj: Load balancing in a distributed environment. In: Todorov, M. (ed.) Applications of Mathematics in Engineering and Economics'34. AIP Conf. Proc., vol. 1067, pp. 527–540. American Institute of Physics, College Park, MD (2008)
29. Drozdowicz, M., Ganzha, M., Paprzycki, M., Olejnik, R., Lirkov, I., Telegin, P., M.Senobari: Parallel, distributed and grid computing for engineering. chap. Ontologies, Agents and the Grid: An Overview, pp. 117–140. Saxe-Coburg Publications, Stirlingshire, UK (2009)
30. Drozdowicz, M., Wasielewska, K., Ganzha, M., Paprzycki, M., Attaui, N., Lirkov, I., Olejnik, R., Petcu, D., Badica, C.: Trends in parallel, distributed, grid and cloud computing for engineering. chap. Ontology for Contract Negotiations in Agent-based Grid Resource Management System. Saxe-Coburg Publications, Stirlingshire, UK (2011)
31. Dynia, M., Korzeniowski, M., Kutyłowski, J.: Competitive maintenance of minimum spanning trees in dynamic graphs. In: Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science. pp. 260–271. SOFSEM '07, Springer-Verlag, Berlin, Heidelberg (2007), `http://dx.doi.org/10.1007/978-3-540-69507-3_21`
32. Ericsson, A.: Erlang/otp system documentation (2010), `http://www.erlang.org/doc/pdf/otp-system-documentation.pdf`
33. Ericsson, A.: Os_mon reference manual. `http://www.erlang.org/doc/apps/os_mon/os_mon.pdf` (2011)
34. Eriksen, M.: Scaling scala at twitter. In: ACM SIGPLAN Commercial Users of Functional Programming. pp. 8:1–8:1. CUFP '10, ACM, New York, NY, USA (2010), `http://doi.acm.org/10.1145/1900160.1900170`
35. Eriksson, K., Williams, M., Armstrong, J.: Program development using erlang - programming rules and conventions. `http://www.erlang.se/doc/programming_rules.pdf` (1996)
36. Foster, I., Jennings, N.R., Kesselman, C.: Brain meets brawn: Why grid and agents need each other. Autonomous Agents and Multiagent Systems, International Joint Conference on 1, 8–15 (2004)
37. Foster, I., Kesselman, C. (eds.): The Grid 2, Second Edition: Blueprint for a New Computing Infrastructure. The Elsevier Series in Grid Computing, Elsevier (2004)

38. Franklin, S., Graesser, A.: Is it an agent, or just a program?: A taxonomy for autonomous agents. pp. 21–35. Springer-Verlag (1996)
39. Galstad, E.: Nagios website. http://www.nagios.org/ (2011)
40. Jennings, N., Wooldridge, M.: Agent technology: foundations, applications, and markets. Springer (1998)
41. Kafura, D., Mukherji, M., Lavender, G.: Act++ 2.0 : A class library for concurrent programming in c++ using actors (1992)
42. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the jvm platform: A comparative analysis. In: PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. pp. 11–20. ACM, New York, NY, USA (2009)
43. Kim, W.: Thal: An actor system for efficient and scalable concurrent computing (1997)
44. Kuranowski, W., Ganzha, M., Gawinecki, M., Paprzycki, M., Lirkov, I., Margenov, S.: Forming and managing agent teams acting as resource brokers in the grid—preliminary considerations. International Journal of Computational Intelligence Research 4(1), 9–16 (2008)
45. Kuranowski, W., Ganzha, M., Paprzycki, M., Lirkov, I.: Supervising agent team an agent-based grid resource brokering system—initial solution. In: Xhafa, F., Barolli, L. (eds.) Proceedings of the Conference on Complex, Intelligent and Software Intensive Systems. pp. 321–326. IEEE CS Press, Los Alamitos, CA (2008)
46. Kuranowski, W., Paprzycki, M., Ganzha, M., Gawinecki, M., Lirkov, I., Margenov, S.: Agents as resource brokers in grids—forming agent teams. In: Proceedings of the LSSC Meeting. vol. 4818, pp. 472–480. Springer, Berlin (2007)
47. Letuchy, E.: Erlang at facebook: Chat architecture. Presented at the Erlang Factory 2009, San Francisco, CA (2009)
48. Makki, S., Havas, G.: Distributed algorithms for depth-first search. Information Processing Letters 60(1), 7–12 (1996)
49. Massie, M.L., Chun, B.N., Culler, D.E.: The ganglia distributed monitoring system: Design, implementation and experience. In: Parallel Computing, vol. 30, pp. 817–840. Elsevier (2004)
50. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Inc, 2 edn. (Jan 2011), http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0981531644
51. Ponci, F., Cristaldi, L., Monti, A., Ottoboni, R.: Multi-agent based power systems monitoring platform: a prototype. In: Power Tech Conference Proceedings, IEEE. vol. 2, p. 5. Bologna, Italy (2003)
52. Ponci, F., Deshmukh, A., Cristaldi, L., Ottoboni, R.: Interface for multi-agent platform systems. In: IEEE-Instrumentation and Measurement Technical Conference. vol. 3, pp. 2226–2230. Ottawa, Canada (2005)
53. Rehak, M., Pechoucek, M., Grill, M., Stiborek, J., Bartos, K., Celeda, P.: Adaptive multiagent system for network traffic monitoring. IEEE Intelligent Systems 24, 16–25 (May 2009)
54. Remy, J., Souza, A., Steger, A.: On an online spanning tree problem in randomly weighted graphs. Combinatorics, Probability and Computing p. 2005 (2005)
55. Ribičre, M., Charlton, P.: Ontology overview. Motorola Labs, Paris (2002), http://www.fipa.org/docs/input/f-in-00045/f-in-00045.pdf
56. Richardson, J.E., Carey, M.J., Schuh, D.T.: The design of the e programming language. ACM Transactions on Programming Languages and Systems 15, 494–534 (1993)

57. Riley, G.: CLIPS website. http://clipsrules.sourceforge.net/ (2011)
58. Santoro, C.: exat: Software agents in erlang. http://www.erlang.org/euc/05/ (2005), december 2011
59. Scheurer, C.A., Scheurer, H.K., Kropf, P.G.: Load balancing driven process migration. Tech. rep., Inst (1995)
60. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for java. In: Proceedings of the 22nd European conference on Object-Oriented Programming. pp. 104–128. ECOOP '08, Springer-Verlag, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-70592-5_6
61. Stefano, A.D., Santoro, C.: Designing Collaborative Agents with eXAT. Enabling Technologies, IEEE International Workshops on pp. 15–20 (2004)
62. Stefano, A.D., Gangemi, F., Santoro, C.: Eresye: an erlang expert system engine. In: Fourth ACM SIGPLAN Erlang Workshop. Tallin, Estony (2005)
63. Stefano, A.D., Santoro, C.: exat: an experimental tool for programming multi-agent systems in erlang. In: AI*IA/Taboo Joint Workshop on Objects and Agents. Villasimius, Italy (2003)
64. Stefano, A.D., Santoro, C.: Building semantic agents in exat. In: WOA. pp. 28–36 (2005)
65. Tismer, C.: Continuations and stackless python. Tech. rep.
66. Tunnell-Jones, A.: dnssd_erlang website. https://github.com/andrewtj/dnssd_erlang
67. Varela, C.A., Agha, G., Wang, W., Desell, T., Maghraoui, K.E., LaPorte, J., Stephens, A.: The SALSA programming language: 1.1.2 release tutorial. Tech. Rep. 07-12, Dept. of Computer Science, R.P.I. (Feb 2007)
68. Vinoski, S.: Concurrency with erlang. IEEE Internet Computing 11(5), 90–93 (2007), http://doi.ieeecomputersociety.org/10.1109/MIC.2007.104
69. Wiger, U.: Four-fold increase in productivity and quality—industrial-strength functional programming in telecom-class products. http://www.erlang.se/publications/Ulf_Wiger.pdf (2001)

Gleb Peregud is an MS student at the Warsaw University of Technology, where he is researching synergies of agent-oriented programming and the actor model, in practical applications. He is an Erlang enthusiast, looking for practical application of actor model and massive concurrency in commercial and academic fields.

Julian Zubek is an MS student at the Warsaw University of Technology in Poland. In his MS thesis, he is developing an experimental Ruby to C compiler. His research interests include also artificial intelligence and programming paradigms (including agent-oriented programming).

Maria Ganzha obtained M.S. and her Ph.D. in Applied Mathematics from the Moscow State University, Moscow, Russia in 1987 and 1991 respectively. Her initial research interests were in the area of differential equations, solving mixed wave equations in space with disappearing obstacles in particular, currently she works in the areas of software engineering, distributed computing and agent systems in particular. She has published more than 100 research papers and is on editorial boards of 5 journals and a book series and was invited to Program Committees of over 100 conferences.

Marcin Paprzycki (Senior Member of the IEEE, Senior Member of the ACM, Senior Fulbright Lecturer, IEEE CS Distinguished Visitor) has received his M.S. Degree in 1986 from Adam Mickiewicz University in Poznan, Poland, his Ph.D. in 1990 from Southern Methodist University in Dallas, Texas and his Doctor of Science Degree from Bulgarian Academy of Sciences in 2008. His initial research interests were in high performance computing and parallel computing, high performance linear algebra in particular. Over time they evolved toward distributed systems and Internet-based computing; in particular, agent systems. He has published more than 350 research papers and was invited to Program Committees of over 400 international conferences. He is on editorial boards of 14 journals and a book series.

Received: January 8, 2012; Accepted: June 7, 2012