International Journal of Energy, Information and Communications
Vol. 2, Issue 4, November, 2011

Persistent Memory in Multi-Agent Systems - The OMAS Approach

Jean-Paul A. Barthès

UMR CNRS 6599, HEUDIASYC,
Université de Technologie de Compiègne, France
{barthes@utc.fr}

Abstract

In long living multi-agent applications the problem of memory becomes important. Agents need to remember what they have done, maintain data persistently despite crashes, power failures, or turning the machines on and off. Other important problems concern the maintenance of agents and the evolution of the applications. In this paper we only consider systems of cognitive hybrid agents capable of multi-processing, and examine the various problems concerning the persistency of information during the agent lifecycle, ranging from keeping a piece of information while executing a specific task to organizing a long term memory for applications that need to run for many months. As an example of such system we take the OMAS platform developed at UTC.

1. Introduction

Agents are often described as persistent software entities [10]. We face here two questions: (i) what is meant by persistence; and (ii) how do available multi-agent platforms support persistence? A quick answer to the second question is that, excepting a few cases concerning mobile agents, they don't. Persistency is usually introduced as the property that an agent possesses allowing it to function as an independent process, with capacities of learning from the interactions with other agents in the system. Many papers have been published dealing with goal persistency or learning, but the model they propose assumes that the agent process runs continuously in a stable environment. In practice agent processes are running on physical machines that may stop for various reasons like crashes or power failures. An agent must thus protect its memory where data and knowledge are being recorded. Most of the projects dealing with intelligent cognitive agents being academic, this is simply not done. One reason is that it makes programming more complex in an already fairly complex world.

A special case pertains to agents on mobile devices for which a break of communication is a serious problem. In that case some sort of persistence must be provided to allow the agents to recover when they are reconnected. However, the agents implied in such environments are lightweight agents [5], which does not concern us in this paper.

The paper examines the different levels requiring storage of information that must be handled by an agent from a development and programming point of view. We call such levels

the agent *memory*. We then will present how the various features can be implemented in a multi-agent system (MAS) platform in order to simplify application programming, using the example of the OMAS platform.

2. The problem of Memory in Complex Multi-Agent Systems

In the rest of the paper we consider systems of cognitive agents. Agents have skills, goals, can undertake several actions at the same time (muti-threading), they have a memory, a model of the environment, ontologies and knowledge bases. We thus do not consider reactive agents nor lightweight mobile agents. We shall also use the terms process and thread interchangeably. Systems in which we are interested comprise for example information systems including personal assistant agents interacting with humans.

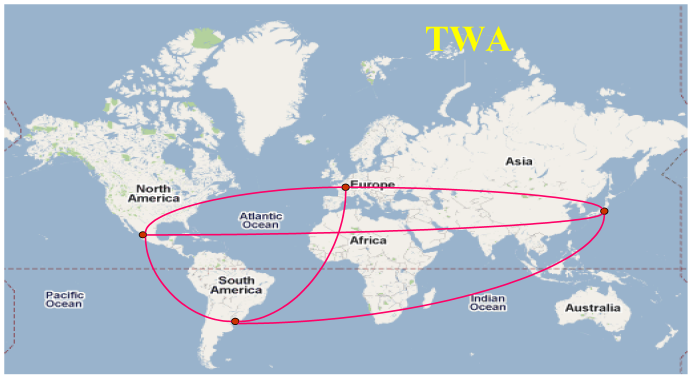

Figure 1. TWA project

To fix ideas, let us introduce an example from an on-going project called TWA (Trans World Agents) linking France, Brazil, Japan and Mexico (Fig.1), in which several humans are connected to a multi-agent system by one or more personal assistants for sharing information. Personal assistants have a local technical agent called NEWS that keeps information and cooperates with the other agents. A global agent named PUBLISHER produces a leaflet summarizing the news classified by categories (Fig.2) every month. The system works as follows: each user provides information that she may keep private. If the information is private, then it is stored locally in her NEWS agent. If the information is public, then it is sent to the PUBLISHER. Each news item is categorized and eventually indexed by keywords. Users have different languages and the PUBLISHER has a multi-lingual ontology. We use this example to illustrate some of the problems that may occur during the functioning of this system. Note that the example is similar to that of a workflow.

2.1. Levels of Memory

Let us consider a simple task like User1 wants to create a new keyword. There is a (short) dialog with PA1 during which PA1 asks NW1, a staff agent, whether the keyword exists or not. If it does not exist then NW1 is going to ask PB, the publisher, to create it.

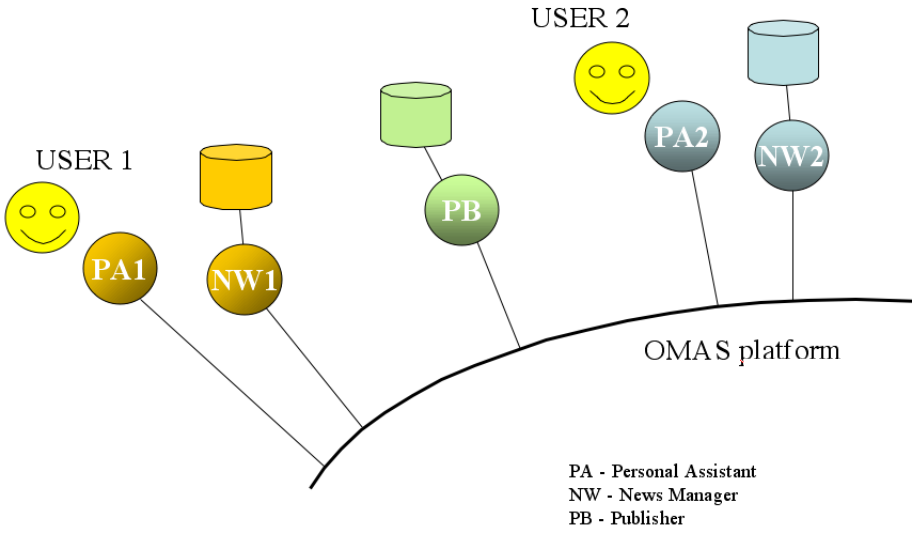

Figure 2. TWA/NEWS architecture

Task Level Let us consider what happens in practice when the keyword does not exist. NW1 is executing a specific task, say T1. Finding that the keyword does not exist results in asking PB, the publisher, to create it. If PB is up and running, then the answer will come back after a delay, waking up the waiting task T1. If PB is not available for some reason and does not answer, one must first detect the condition, and then do something about it. Detecting the condition is done by specifying a timeout and associating a timeout handler. Then, when it is found that PB is not available one can periodically send a request until PB comes back up in line again (polling). In that case, even if T1 and the timeout handler execute in different threads, all data must be saved in the context of the task T1, and not be mixed up with other tasks executing in parallel.

Agent Level Now another solution can be selected. Instead of polling the publisher PB, one can have PB broadcast a message when it comes back up on line, advertising it and asking if there are some deferred actions to execute. In that case, a temporary structure must be set up by NW1, containing the required information for creating the keyword. There is no interest in keeping the task T1 active any longer, and because T1 will terminate, information has to be recorded at the agent level. In the same fashion, information concerning the agent environment or the model of its mood, being independent from the tasks that created or modified them, must be stored at the agent level. Usually however, this information is only valid during the length of a session and can be discarded at the end of a session.

Dialog Level When the agent is a personal assistant, the information it handles during the conversation with its master must be saved somewhere, at least during a conversation turn. The best place to save it is in the dialog process.

Long Term Data and Knowledge Finally, when an agent is learning either a strategy or the preferences of its master, it is advisable to keep the resulting data structure in a longer perspective, sometimes over many sessions. Thus, one needs a persistent store, either a data base or an object store. Furthermore, the saved data or knowledge must be structured in such a way as to be efficiently and easily retrieved when needed.

Table 1 summarizes the different levels of persistency for the different types of data handled by an agent.

Table 1. Different levels of data persistency

Level	Duration	Concerned data
task	data last the length of a task, normally at most a few seconds, rarely more	data used among the different processes attached to a particular task. Data are protected across tasks.
agent	typically the length of a session, i.e. minutes to hours	data are saved at the agent level and are shared among all the tasks and processes associated with the agent.
dialog	usually the length of a dialog turn	data associated with a particular dialog. Contains information to interpret the dialog and parameter values concerning the goal of the dialog.
long term	long lasting data (months)	concern ontological information, knowledge bases, various representations, various profiles

2.2. Requirements in the Life Cycle of a MAS Project

During the life cycle of a MAS project one can distinguish four phases: the development phase, the deployment phase, the maintenance phase, and the evolution phase. Each phase has its own requirements regarding long term persistency.

Application Development During the development of the application, the code will be modified frequently and tests will be run, restarting everything from scratch. Consequently, persistency is more of a hindrance as it is a help. Persistency should be turned off. In many situations, e.g. when using a database, one can simply erase the content of the database, or the whole database and restart the application, recreating the necessary persistent information.

Application Deployment During the deployment phase, one must be more careful, because persistent data are partly the result of agent interaction that already occured. An agent can be modified, by adding or removing skills or goals. When an agent is loaded some representation data is produced (e.g. the description of the agent and of its main features). Such data must not be in conflict with the data saved in the persistent store. Or, at least, they should replace the previous data. Depending on the representation formalism, this is not always easy to do.

Application Maintenance During the life of the application, there may be times where the supporting platform or representation system undergoes significant changes. The situation is not too bad when new features are added. It is worse when something is removed.

In that case, the new platform should be upgraded without damage to the application and to the long term data (e.g. when there is an evolution of the representation formalism).

Application Evolution A last case concerns the evolution of the application. It is done by adding or removing agents, adding new skills, goals, concepts, properties, changing interfaces, adding new features. We are basically in the same situation as in the deployment phase, and would like to proceed without stopping the application. In such a case the structure of the representation models (concepts) might be changed significantly.

2.3. Proposed Solutions

Most of the proposed solutions to the persistency problem use standard software approaches.

Regarding persistency at the task level or session level, the solutions depend on the type of threads used to support multi-processing. It the threads are light threads running in the same environment, then the solution consists in using shared global variables, being careful about possible interactions. It the threads are system threads, meaning that each thread is independent, then the solution may be more complex depending on the supporting environment and language.

An interesting solution proposed in the 80s by object databases is to use a persistent language [2, 8]. In such languages, persistency is attached to classes (e.g. ontology concepts) and instances are saved automatically, the job being done by the compiler. The approach has however some drawbacks: not everything that needs to be saved is saved; when a machine crashes the system may end up being broken if cache pages have not been rewritten back to disk.

Another approach is to use checkpointing, which in the context of agents amounts to suspending agents. The approach is proposed by Motomura et al. [13] in the Maglog platform. An agent can be suspended, its state is saved, the machine can be stopped and the agent is resumed later when the machine is restarted.

A different approach to ensure against the failure of a broker agent has been proposed by Kumar et al. [11]. It consists in restructuring the set of agents and allowing the creation of new brokers dynamically.

Finally, traditional transaction techniques can be extended to the agent world as demonstrated by Martinez and Alvarado [12]. However, their approach is better suited to mobile computing.

Next section discusses how the different levels of memory have been implemented in the OMAS platform.

3. Overview of the OMAS Platform

The OMAS platform is a platform hosting hybrid agents that can be programmed to be cognitive. It is used in a research context for developing application prototypes. In the development of an application both the application and the platform may change. In particular the platform may be enriched by new features required by the application. The platform was designed and first implemented in the late 80s.

We first introduce briefly the OMAS platform. It is detailed in Barthès [3] or in its documentation[1].

3.1. OMAS Architecture

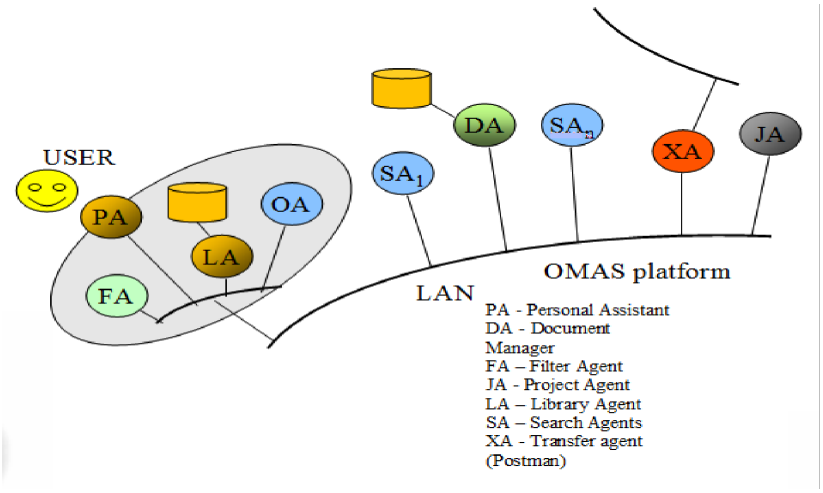

Figure 3. Coterie architecture showing Personal Assistant and staff agents (FA, LA and OA) on a local coterie, and Transfer Agent XA connecting another remote local coterie

OMAS organizes agents in closely knitted groups called *local coteries* located on a single network loop (Fig.3). Agents in a local coterie are of three types: (i) service agents (SA); (ii) personal assistants (PA); and transfer agents (XA). Service agents provide various types of services depending on the application, personal assistants are used to interface humans, and transfer agents are used to connect other coteries or remote systems or other multi-agent platforms. A *coterie* is a set of local coteries connected through transfer agents. In a coterie all agents receive all messages, so that each agent can decide to exploit or not a message content, even though it is not the intended receiver of the message. Agents are like persons located in the same room overhearing everything said in the room. An OMAS agent has a complex structure shown Fig.4. An OMAS agent typically runs 6 to 20 threads (processes) in parallel.

A personal assistant is an agent in charge of interfacing a particular person. It is associated with this person, its master, and tries to simplify the interface with the multi-agent system. A PA has a default interaction window. To reduce the complexity of a PA and to allow evolution, a PA has a limited ontology and knowledge base and relies on a set of staff assistants that are service agents specialized in some types of tasks or services. This approach implements the concept of *Digital Butler* proposed by Negroponte [14].

3.2. Service Agent Structure

To understand the problem of the storage of data and information we need to introduce part if the internal organization of an agent, namely how the requests are processed, and

[1]Available at www.utc.fr/~barthes/OMAS/

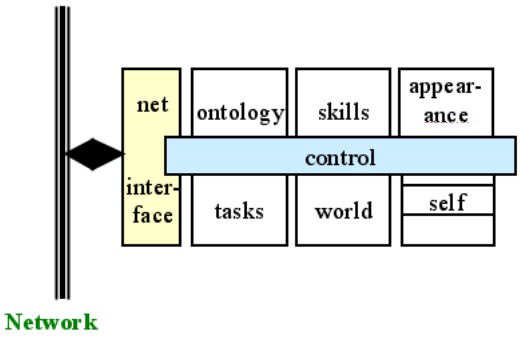

Figure 4. Generic structured of a service agent

also how the ontology and knowledge base are organized.

3.2.1. Processing a Message

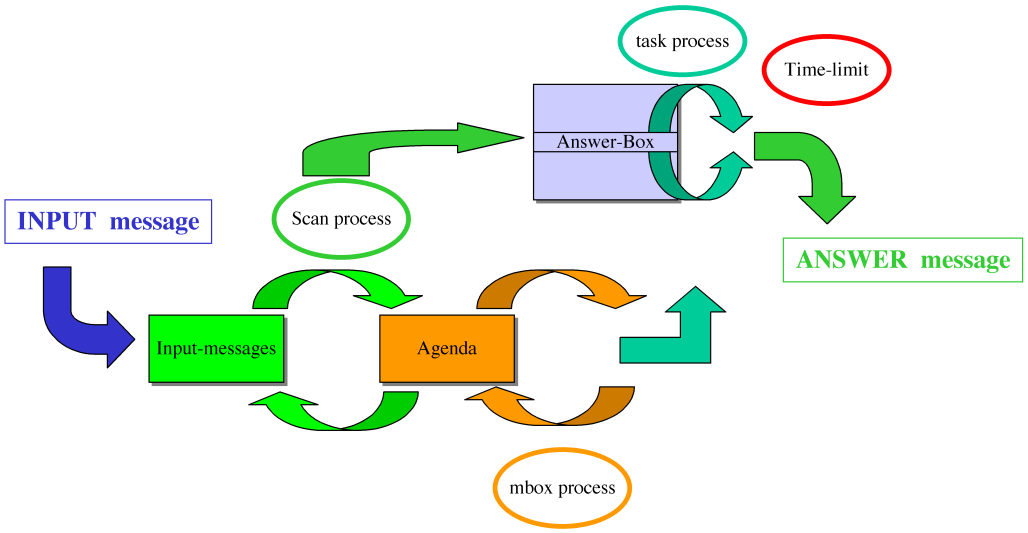

Figure 5. Agent processes

It is important to understand how a request is processed, because data has to be kept and moved around during the execution of a task answering the request. An OMAS agent runs several processes as shown Fig.5. When a message is received into its mailbox, then, if it calls for a reactive answer it is executed readily by the scan process watching the mailbox. If it is a request it is inserted into an agenda. If it is an answer, it is routed to the proper task thread. A second process called main watches the agenda and as soon as a request is inserted creates a task to serve it with two different threads, a thread for processing the task and a time-limit thread to kill the task after some times if it stays hanging out (default time limit is 1 hour). Requests are executed by skills typically defined as follows:

```
(defskill :buy-book
   :static-fcn <static function>
   {:dynamic-fcn <dynamic function>
```

```
{:timeout-handler <user timeout handler>}})
```

where brackets indicate an optional argument.

When a request is processed, first the static part of the skill in invoked, and, if the task does not call for the help of other agents, it returns the answer or else commits suicide and does not answer. If the task needs to create subtasks, then messages are sent to other agents and the agent waits on a second part of the skill, called the dynamic part of the skill, in the same process. If in addition a timeout is specified a timeout process is created to call an eventual timeout handler when the time expires.

Thus, when a service agent executes `n` requests concurrently the number of involved processes is

```
N = 2 + (n * p) + q
```

where `n` is the number of parallel requests, `p` is normally 2 (3 if a timeout has been set up), and `q` is the number of additional processes set up by each executing skill. Each request however corresponds to a task referenced in all processes related to this task. The organization and control of all processes are performed by the OMAS platform.

3.2.2. Ontology and Knowledge Base Formalism

The agent long term memory will contain its ontology and knowledge base. Thus, it is important to look at the way they are structured.

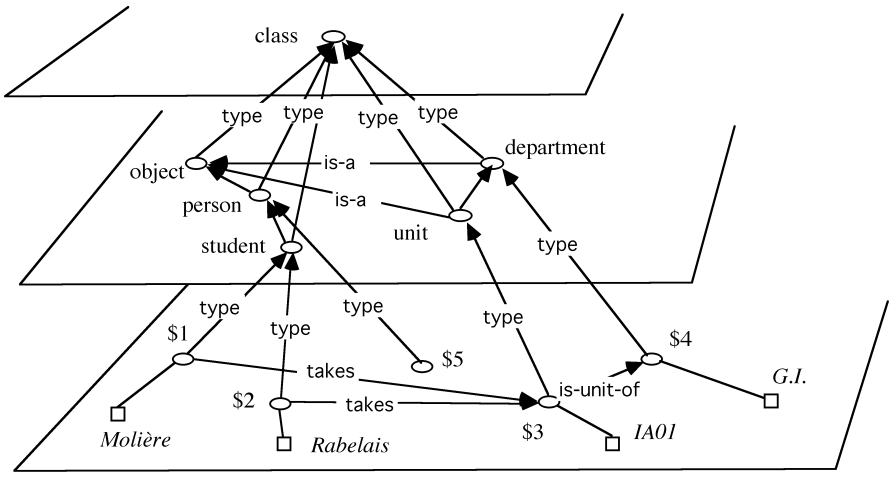

Figure 6. The PDM/MOSS structural abstraction hierarchy

The representation in OMAS is implemented through a frame based semantic net formalism called PDM/MOSS that was developed during the 70s. It is a reflexive representation in which classes (concepts) and metaclasses are first order objects. It contains an object-oriented mechanism and methods are also first class objects. In addition, it contains classless objects, virtual classes and virtual properties, different types of methods. The structure of the representation is shown Fig.6. More information can be found in [3] or in the web site[2]. The representation emphasizes typicality rather than prescription, insisting on default mechanisms.

[2]www.utc;fr/∼barthes/MOSS/

The intentional representation of ontologies correspond to the middle plane of Fig.6, the knowledge base containing individuals is situated at the lower level. The top level shows an object that is its own representation (reflexivity). Inheritance can occur at both levels.

Because of its flexibility the representation system has been integrated into the OMAS platform and allows representing ontologies and knowledge bases easily and uniformly. Reasoning is provided by methods and by a query language, all integrated in the environment. References to the different objects represented in the agent memory are also used in the platform content language.

Let us take an example of the creation of a new concept. Table 2 shows the creation of the concept of *Contact*. It has a few attributes and relations, as well as restrictions on the *type* attribute and an indexable attribute *ref*. When creating the concept a number of ancillary objects are created as well as functions (methods) that will be invoked when creating instances, in the tradition of frame languages found in artificial intelligence. Furthermore, links are made with the other concepts mentioned in the relations. Since an agent may create new concepts at any time it is necessary to save all the resulting objects and functions if the agent has a long term memory.

Table 2. The concept of contact

(defconcept	”contact”
	(:att ”ref” (:entry))
	(:att ”contact type” (:one-of :occasional :permanent))
	(:att ”start date” (:unique))
	(:rel ”country” (:to ”country”))
	(:rel ”foreign correspondent” (:to ”person”))
	(:rel ”partner” (:to ”organization”))
	(:rel ”UTC correspondent” (:to ”person”))
	(:att ”comment”))

4. OMAS Memory Levels

In this section we try to show how the different problems of persistency were solved in the OMAS platform. Since we are mostly developing prototypes, we try to simplify programming. In all cases we tried to provide functions that hide the complexity of the background mechanisms so that using the different levels of memory does not require complex programming.

4.1. Task level

When executing several functions in the same thread, like a static-skill and a dynamic skill function, one can use a storage area attached to the task itself, using two functions: env-set and env-get.

For example, if we set the following text value in the static function of a skill

```
(env-set agent "...some text..." :text)
```

It can be retrieved later, e.g. in the dynamic function of a skill by calling

```
(env-get agent :text)
```

It can also be retrieved in the same fashion while executing a user-defined timeout handler, although this handler does not execute in the same process.

If two requests are processed in parallel there is no confusion between the stored values, because they are stored at task level and not shared among threads that are not part of the same task.

An additional advantage of this approach is that when the task is terminated, the associated storage area is automatically garbage-collected.

4.2. Agent Level

When the information has to be accessed from different threads, it is necessary to associate it with the agent itself since the agent is seen by all its threads. In such a case data are stored in memory items in the short term memory of the agent. The short term memory is managed by three functions: remember, recall and forget. For example:

```
(remember agent "...some text..." :text)
```

and later in another function or another thread

```
(recall agent :text)
(forget agent :text)
```

will retrieve the text and clean the memory.

In that case the information is shared by all the threads and all the skills since it is recorded at the agent level. If two requests require the same skill and execute in parallel, then there might be inconsistencies in using this mechanism.

An advantage of this approach is that the information may be shared by several threads. A disadvantage is that one must be careful, and that one must clean the memory when the information is no longer required.

Table 3. Structure of a memory element

Property	meaning
date	date of creation of the element (assigned by the system)
tag	a symbol used by the remember, recall and forget functions to access the element (usually a keyword)
service	the name of a service associated with the element (user definable)
info-type	type of information (user definable)
ontology	ontology associated with the information (user definable)
content	content (value) of the memory element

A memory element is structured and may have different properties listed in Table 3.

4.3. Dialog Level

A Personal assistant is involved in dialogs with the user, its master. Dialogs are run in specific threads. An OMAS dialog is intended to end up with an action (embodied by a message being sent somewhere). Dialogs are described by conversation graphs (finite state machines) that are traversed according to the answers from the master and the state of the environment. Transition rules act on a specific fact base that is attached to a conversation object containing other information like the I/O channels or the agent name. The dialog

mechanism is described elsewhere [4]. Some information related to the dialog is transient and can be discarded when the goal has been reached. Other information must be kept in the long term. Again, like in the previous cases a specific area is devoted to storing the transient information, and can be accessed with several functions like set-fact and get-fact. A personal assistant can handle several conversations in parallel. Each conversation will have its own memory area like in the task case. Of course, in addition, information can be shared by using the agent short term memory.

4.4. Long Term Data and Knowledge

Long term data consist of permanent data associated with a particular agent, e.g. an ontology, a knowledge base, various representations of the environment, various profiles. They must be kept over a long time and must survive sessions and crashes. Among OMAS agents only service agents and staff agents can have persistent memory. This decision was made due to the complexity of handling persistency with other agents.

To declare an agent persistent, meaning that it will have a persistent long term store, one does it simply when creating the agent, for example

```
(defagent :publisher :persistency t)
```

The result of this declaration is that the agent is given a permanent object store and its ontology and KB will be installed into this store the first time the agent is loaded, from its ontology text file. Afterwards, when the agent is restarted (reloaded) it will bypass the text files and connect with the object store directly.

The persistent store is transparent to the objects using the PDM/MOSS representation, meaning that a referenced object will be loaded automatically when used, if is is not in memory. This resembles the behavior obtained with a persistent language. Changes however have to be done during transactions, which are specified by using macros or functions listed in Table 4.

Table 4. Functions implementing transactions

Property	meaning
with-transaction body	body (list of instructions) is executed in the context of a transaction
agent-start-changes agent	declares the beginning of a transaction
agent-commit-changes agent	declares the end of a transaction by committing it
agent-abort-changes agent	declares the end of a transaction by aborting it

During a declared transaction all changes to PDM/MOSS objects will be saved or discarded at the end of the transaction automatically.

Although the approach looks simple and easy, there are some problems to be described in the next section. In addition, for an agent, one would like to have a better structured long term memory than a simple frame based semantic net. The latter point is addressed in Section 6.

5. OMAS Approach to Agent Life Cycle Problems

Because OMAS is a living platform it has to face the problems mentioned in Section 2.2, mostly due to the long term memory. An additional difficulty is linked to the rich structure of the semantic representation that we want to keep in the agent memory.

Development During the development phase of a prototyped application, many changes have to be done and the prototype is constantly restarted. In this context it is better to turn the long term memory off, which with OMAS can be done very simply by setting the persistency option in the agent definition to nil. In that case everything is created each time from text files.

Deployment The deployment phase is a delicate phase, since long term memory starts having information, but there are still some bugs in the application code. Data kept at task level of agent level do not pose any problem and can be recreated when the application is restarted, but the long term memory has to be managed carefully. We can do this by using checkpointing, meaning that at some stage the content of the memory is saved, or exported into a text file, kept, then reloaded later on if there is a problem. If worse comes to worst, one can export the memory into a human readable text file, which allows correcting it manually. The database is then erased and recreated by reloading the agent text files.

Maintenance Maintenance is a different problem. We want to keep the application as it is but upload new versions of the underlying platform. This is in practice impossible if all the representational metadata and methods are stored on disk interleaved with the application data. Thus, the solution adopted by OMAS is to carefully separate metadata objects and all the objects related to the agents from the application objects. When the application is restarted, all metadata objects and objects directly related to agent descriptions are recreated and are reconciled with the data kept in the long term store. Although it is a non trivial process, the work is done by the platform and the application developer has nothing special to do.

Finally we should mention that for efficient accesses we do not use relational databases, because in a relational database the schema is static and cannot be modified easily. We use object stores that look like gigantic hash tables containing objects that can change dynamically.

Application Evolution Application evolution means that we want to add or remove agents and features to the application. This in general does not pose difficult problems if the changes are not drastic. Adding new agents does not change the long term memories of the existing agents, and ontologies are personal to the agents. In OMAS, the only part to modify are the dialogs of the personal assistants, which is easy because those are kept in text files since they need to be adjusted constantly.

6. Conclusion

In summary we considered the problem of persistency of the data and showed that there are several levels for managing the data according to their usage. The long term level is the

most difficult because it implies saving information in a persistent store capable of handling dynamic extensions. Many problems can then occur during the agent life cycle that may introduce inconsistencies. We showed how the OMAS platform handles the problem given that the goal is to preserve the structure of the ontology and knowledge base as a frame-based semantic net.

Related Work We could not find many platforms that address the problem of data and knowledge persistency. Some authors write about goal persistency, but they assume that the agents are executing in a stable environment without crashes or power failures.

Some work has been done for the control of manufacturing processes by agents as mentioned by Heikkilä et al [9]. Persistency has also been studied for groups of reactive agents by Choi et al [7].

The problem of persistency is popular in the context of mobile agents, since communications can be broken easily. Fault tolerance architectures are offered, for example by Bajwa et al. [5] who rely on persistent objects. Another approach proposed by Kumar et al. [11] advises to keep a sufficient number of broker agents alive in the system.

Athanasiadis et al. [1] offer building blocks for building agent systems with possible connections to relational databases through the use of object-relational mappings (Hibernate). We already stressed the static character of relational databases, which makes them unsuitable to our purposes.

As mentioned previously, Motomura et al. [13] developed a platform, Maglog, allowing agents to be suspended. The platform keeps the state of the agent and corresponding fields. Agents can then be restarted when the server resumes. The approach is somewhat analogous to checkpointing, since saving the state of an agent is a synchronous decision.

The JADE platform[3] implements persistent objects as an add on to the previous versions. The approach looks quite similar on the surface, since objects may be declared as persistent, and changes are recorded when they are done within a transaction. The approach is however at a much lower level than in OMAS. The main difference is in the rich PDM/MOSS representation of OMAS objects. When creating a PDM/MOSS object, a number of auxiliary objects and functions (methods) are created at the same time, especially when creating a new concept, and the new object may be linked to a number of other objects. When modifying or deleting the objects all the structures have to be updated, which is done automatically. Therefore, the OMAS store is not a simple persistent cache, but contains structured objects and functions. As a drawback a reconciliation mechanism had to be developed to be run when an agent is reloaded.

Future Developments By working on prototyping many applications, we found that the long term memory must be structured in a better format than frame based semantic nets. K. Chen developed an approach for automatically parsing dialogs and producing cases to be kept as a case base [6]. We tested the approach in the OMAS platform, but it needs yet to be integrated.

The OMAS platform is available at www.utc.fr/∼barthes/OMAS/.

[3]http://jade.tilab.com/

References

[1] Ioannis Athanasiadis, Ferdinando Villa, Andreas-Emilio Rizzoli. Ontologies, Javabeans and Relational Databases for Enabling Semantic Programming. Proc. 31st International Computer Software and Applications Conference, Beijing, China, Vol. 2, pp. 341-346, 2007.

[2] François Bancilhon, Claude Delobel, Paris Kanelakis. Building an Object-Oriented Database System - The story of O2. Morgan-Kaufmann, San Mateo, CA, 1992.

[3] Jean-Paul A. Barthès. OMAS - A flexible multi-agent environment for CSCWD *Future Generation Computer Systems*, Vol. 27, pp. 78-87, 2011.

[4] Jean-Paul A. Barthès. Flexible Communication Based on Linguistic and Ontological Cues *E-Technologies: Transformation in a Connected World, Lecture Notes in Business Information Processing, Springer*, Vol. 78 Part 4, pp. 131-145, 2011.

[5] Aqsa Bajwa, Sana Farooq, Obaid Malik, Sana Khalique, Hiroki Suguri, Hafiz Farooq Ahmad, Arshad Ali. Persistent Architecture for Context Aware Lightweight Multi-agent Systems. in Programming Agents, Lecture Notes in Computer Science, vol. 4411/2007, pp. 57-69, 2007.

[6] Kejia Chen, Jean-Paul A. Barthès. Structuring a Case-Based Memory for Personal Assistant Agents. *International Journal of Cognitive Informatics and Natural Intelligence*, Vol. 4, No. 1, pp. 45-64, 2010.

[7] Dongkyu Choi, M. Kaufman, P. Langley, N. Nejati, D. Shapiro. An Architecture for Persistent Reactive Behavior. Proc. Third Intl. Joint Conference on Autonomous Agents and Multi-Agent Systems, pp. 988-995, 2004.

[8] Eric N. Hanson, Tina M. Harvey, Mark A. Roth. Experiences in Database System Implementation Using a Persistent Programming Language. *Software - Practice and Experience*, Vol. 23, No. 12, December 1993.

[9] Taio Heikkilä, Martin Kollingbaum, Paul Valckenaers, Geert-Jan Bluemink. An agent Architecture for Manufacturing Control: manAge. *Computers in Industry*, Vol. 46, No. 3, pp. 315-331, October 2001.

[10] Nick Jennings, Michael Wooldridge. Software Agents. *IEEE Review*, pp. 17-20, January 1996.

[11] S. Kumar, P.R. Cohen, H.J. Levesque. The Adaptive Agent Architecture: Achieving Fault Tolerance using Persistent Broker Teams. Proc. Intl. Conference on Multi-Agent Systems, Boston, MA, pp. 159-166, 2000.

[12] Jorge Martinez, Matias Alvarado. Concurrency Control Monitor for Nested Transactions Based on Autonomous Agents. *Revista Iberamericana de Inteligencia Artificial*, Vol. 9, No. 25, pp. 29-38, 2005

[13] S. Motomura, J. Kishida, T. Kawamura, K. Sugahara. Realization of Persistency in a Multi-Agent Framework. Proc. of Intl. Conference on Integration of Knowledge Intensive Multi-Agent Systems, KIMAS 2007, Waltham, MA, pages 28-33, 2007.

[14] N. Negroponte. Agents: From direct manipulation to delegation. In J M Bradshaw, editor, *Software Agents*, pages 57–66. MIT Press, Cambridge, 1997.

Author

Jean-Paul Barthès obtained a PhD from Stanford University in 1973. Currently, he is professor emeritus in the department of Computer Science at the University of Technology of Compiègne (UTC) in France. His main reserach interests are related to Distributed Artificial Intelligence and mixed societies of cognitive artificial and human agents with a focus on Personal Assistant Agents. Prof. Barthès is co-chair of the IEEE-SMC Technical Committee on CSCWD (Computer-Supported Cooperative Work in Design). Under his supervision a number of multi-agent applications have been developed during the last 25 years.