

# **AMETAS**

*Good Migrations.*

## ***AMETAS White Paper Series***

by Klaus Herrmann  
and Michael Zapf

# **On Agents**

July 2000

Johann Wolfgang Goethe-Universität Frankfurt/Main, Germany  
Department of Computer Science

[www.ametas.de](http://www.ametas.de)

## The Asynchronous Message Transfer Agent System

We will start with a brief overview on the basic principles of the agent system AMETAS. The question “what is an agent” has not been answered in a consistent way by the community of agent researchers. Therefore, it is essential that you understand what we define as crucial aspects of agenthood and how this is realized in AMETAS.

### What Is an Agent?

If you don't know what an agent is, you are advised to read the following text carefully. If you think you know what an agent is, you are advised to read the following text carefully nevertheless.

The question “what is agenthood” has been under discussion ever since the topic arose in the mid-1990s. With the upcoming programming language Java, more and more research groups started to develop new ideas on how to use the platform independence and the code migration to solve problems concerning network bandwidth preservation, off-line computing, and so on. Up to then, agents were mostly a topic in (*Distributed*) *Artificial Intelligence*. Users were given electronic assistants which could e.g. survey the user's learning process or build timetables and make appointments.

The idea behind all that can be described as *detaching a program from the direct influence of the user and having it run in the background*. Unlike background processes (known, for instance, from Unix), agents should be flexible, reacting to stimuli and possibly learning. DAI researchers mainly address problems like trying to map the outside world into the agent's knowledge base in order to deduce strategies to solve problems. Another focus is to find flexible ways of communication, for example by KQML, the *Knowledge Query and Manipulation Language*. One definition of agenthood coming from this area of research states that any entity be an agent when it communicates by an agent communication language, which, for example, happens to be KQML.

The most prominent paradigm applied to these agents is abbreviated by BGI, standing for *Beliefs, Goals, and Intentions*. Agents that comply to this paradigm are designed

- to have some sort of knowledge base where they store acquired and predefined facts (*beliefs*);
- to try to solve a set of tasks, either by complex deductive processes or just by simply working down a list (*goals*);
- to decide on actions autonomously that help to solve the tasks (*intentions*).

A lot of theorems and deductions of these aspects led to some definitions of agenthood that were accepted by many researchers.

As mentioned, Java had its breakthrough in the mid-nineties. As a language originally designed for set-top boxes but then re-targetted towards the World Wide Web, Java has a lot of capabilities that facilitate the construction of objects that can be sent through the network. At first, Java was used to write the so-called *Applets* which are programs running on the Virtual Machine defined by Java. Any host on the network running such a VM is able to execute these Applets so different hosts in different networks were enabled to download code without bothering about a required platform for execution.

A natural step was not only offering code as applets to be downloaded by a host but sending code to remote hosts to have it do something at that location. These wandering

programs were called *mobile agents*, in contrast to the agents known before that resided on a node to wait for commands to execute (e.g. SNMP agents that are queried by a SNMP manager).

The problem the researchers faced was that there were several definitions of agenthood that were hard to match — if at all possible. Henceforth, agents were separated in two classes:

- *intelligent agents* as products of DAI research, intended to act as autonomous components of systems trying to provide intelligent behavior;
- *mobile agents* as products of operating systems and distributed systems research, intended to wander from network node to network node trying to solve problems remotely.

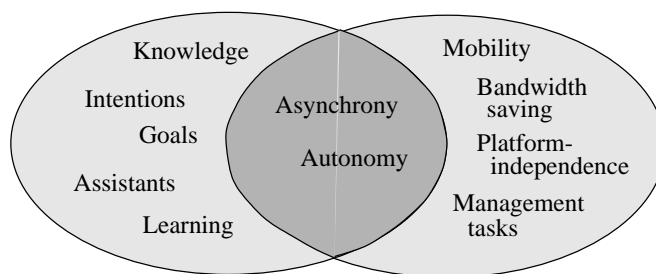


Figure 1: Realms of agenthood

Figure 1 shows the two areas of agent research as being quite different with only little common aspects. Taking a closer look, the areas are actually not that far apart. Aspects being shared by both approaches are that an agent is expected to act somehow *autonomously*, that is, it does not require to be instructed by an external entity like the user to decide on further actions. Even though goals were introduced by the intelligent agent theory, mobile agents also have their own goals, represented implicitly by their implementation. The mobile agent solves its tasks by wandering (or *migrating*) from node to node; intelligent agents process incoming information and try to extract facts. These strategies are not exclusive; a mobile agent also has to gather external information. On the other hand, there is no restriction for intelligent agents that prevents them from migrating if they intend to do it.

While intelligent behaviour is a matter of programming techniques, aspects like migration must be provided by the infrastructure of the agent system. This is due to the fact that currently, no widely used operating system supports the notion of roaming programs. The agent system infrastructure therefore has to provide mechanism for the agent to switch its location.

### Agents from the AMETAS Point of View

We consider AMETAS, for its support of mobility, to belong to the group of mobile agent systems. A difference to many other systems, however, is the emphasis on *autonomy* of the agent. We identified this principle to be the crucial one for qualifying objects to be agents.

Agents in AMETAS are therefore objects that need to be designed to have a complete, own thread of actions. Actually, the environment (the *place*) where the agent is

executed offers a unique thread for each agent. Agents have no possibility to address any other agent directly. When agents want to send information to another agent, they have to use a special facility of the place (*Post Office*) which delivers the message to the *mailbox* of the receiver. However, the sender cannot rely on the receiver to actually process the message. The receiver can decide to postpone the message or to even ignore it — according to its own goals. To put it in a nutshell:

AMETAS agents adopt a client role throughout their existence.

*But I want the agent to do something for me*, you say. Of course, this does *not* break this rule. Just like you can ask any person to do something for you, you can view the agent as a partner that helps you do something. Just as you have your priorities, so has the agent. But you cannot grab the agent and directly modify its internal state by setting internal variables directly or indirectly. We consider this to be such a bad violation of the concept of autonomy that we designed the whole system to prevent this. On the other hand, this restriction to remain in a client role allows a better understanding and design of agent-based algorithms because you can always find analogies between autonomous agents and real persons

## Migration Techniques

Whenever agents decide to migrate, we call this *active migration*. The counterpart is *passive migration* which is used to shift agents from one node to another one. This allows to balance the load on a set of servers. However, passive migration and the autonomy of an agent do not go together: The agent is relocated against its will, so to say. It has to check where it is executed now and probably if all the objects it expects to be present can be found at the new location. Normally, the agent communicates with other agents which would have to be relocated, too. Thus, AMETAS does not support passive migration.

Another important question that must be dealt with is what happens *after* the migration. If the agent decides to migrate actively, it executes some special command like *go* or triggers its transport by sending a special message to the place. We are talking about *strong migration* when at the new location the execution of code continues after the execution of the command that caused the migration. From the point of view of programming logic, the switch of the execution platform is transparent. Otherwise we call it *weak migration*. Normally, the agent is restarted at some well-defined position in its code, for example by calling a certain method of the agent.

AMETAS only supports *active weak migration*. This means that agents have to decide on their own when and where to migrate, and they are restarted after they reach the target place. First, this can be implemented much easier and more efficient. Second, agents that change their execution environment cannot continue with their code as if nothing happened. Strong migration seems to suggest that exactly this were possible. But in practise, the agent has to look for the new communication partners and probably must adjust its plans. We believe that this re-orientation in the new environment is such an important change that it is reasonable to restart the agent.

## How Do Agents Get in Contact with their Environment?

In AMETAS, agents cannot get in direct contact with system resources. This task is done by special objects called *services* and *user adapters*. Services can be seen as ex-

tensions of a place. They are stationary and may even contain native code. Services can be optimized to the conditions of the local place. Figure 2 shows the basic components.

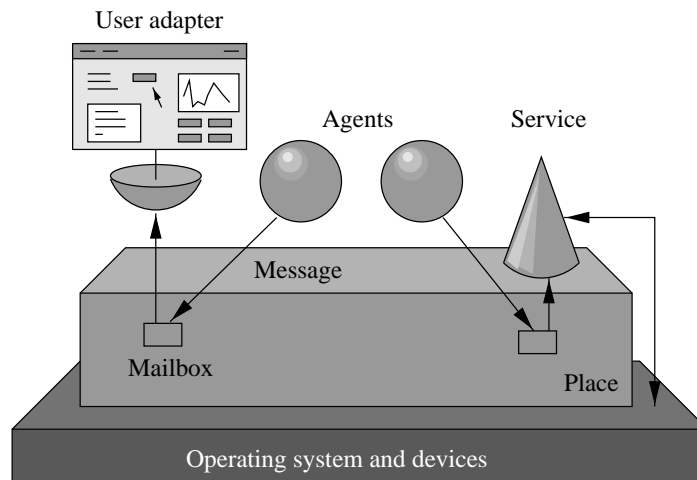


Figure 2: Basic structure of AMETAS

User adapters serve to integrate human users into the agent systems. They may receive keyboard and mouse input and use graphic displays to present data to the user. User adapters are stationary as well and normally only used at the place where the respective user interacts with an agent application.

User adapters, services, and agents all communicate by the same asynchronous message transfer system that is described in more detail in another paper of this series. It is important to note that all of the three components treat any other component equally. That means that when an agent submits its results to the user, it actually *believes* this user to be another agent. These three categories of communicating entities are commonly called *Place Users*. For the sake of brevity and understandability, we sometimes only speak of communication between agents. All communication issues apply to all kinds of Place Users as far as the message exchange via the place is concerned.

If agents should be used to perform some administrative tasks in system management, they need access to low-level resources as, for instance, the file system. This is not possible for the agent alone so it is necessary to design a special service that does the job on behalf of the agent. Whenever migration is not needed, services may be preferable to implement because they have more capabilities. Otherwise, agents need to be used because they are the only Place Users that are able to migrate.

## How to Build up An Agent System

In this AMETAS White Paper Series you will find some useful articles on how to successfully design an agent-based application. We summarize the most important hints for a brief overview:

1. Consider how *distributed* your application should be. Hosting a place requires quite some performance and memory resources from the system. Does every host need an agent place or can you design your application using only few places in each subnet?

2. Do you want to have an *open system* that should be reachable for outside agents, or do you want to have a *closed system* with only your own agents running?
3. If you want to build up an open system you have to determine which domain your system should be part of. (Cf. the article about place name for further details.) Propagate your domain to all interested people.
4. Designing an agent-based application, consider carefully which components will be mobile, which ones will interact with the user and which ones will be used to access system resources. Remember that there is no component that is capable of doing all these tasks.
5. Do you want to hide the agent nature of your application or do you want to keep the user aware of it? You can realize both approaches by implementing suitable user adapters.