

AMETAS

Good Migrations.

AMETAS White Paper Series

by Klaus Herrmann
and Michael Zapf

On Communication

July 2000

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

www.ametas.de

Communication in AMETAS

AMETAS has a very specific view on inter-agent communication. This is already indicated by the name AMETAS which expands to *Asynchronous MESSage Transfer Agent System*. Communication between mobile entities poses entirely different problems than communication between ordinary objects that cannot leave their host. Therefore, it is not advisable to unreflectedly transfer communication paradigms from distributed object systems to mobile agent systems. This White Paper discusses the problems of this transfer and gives a technical overview on the communicational concepts which embody the foundation of AMETAS¹.

Communicational Concepts in Agent Systems

By communication we mean *inter-agent communication*, i.e. the process of two agents exchanging information. Note that when we say “agents”, we mean *mobile agents* by default. This is very important in this context since mobility is a major issue when it comes to agent communication, as we will see later on. Several techniques for agent communication have been proposed. The most popular one is RPC-like communication between mobile agents. Within this communication paradigm, one agent calls the methods of another agent using some standardized communication mechanism like CORBA or RMI². In most cases this is even done if the two agents run in the same agent environment³. The reasons for using RPC-like communication are obvious:

- It is a well-known paradigm for programming distributed systems.
- It is convenient since calling methods is a well-known abstraction in classical paradigm.
- It allows the interoperation with other services based on the same middleware.

The first serious effort to introduce a standard for mobile agent systems was MASIF (or MAF as it was called first). MASIF defines an agent to be a CORBA-object providing several methods for management purposes.

The Drawbacks of RPC-Like Communication

At first glance, method calls seem to be suitable for the mobile agent paradigm. The question arises:

Why could RPC-like communication be considered disadvantageous in mobile agent systems?

¹We will mainly consider agents as communication partners neglecting the inherent differences to the other AMETAS Place User categories (Services and Use Adapters) wherever appropriate. All three Place User types reside on the same communication layer, thus using the same mechanisms to communicate with each other.

²Note that we are very well aware of the fact that RMI is not a middleware platform since it only supports communication between Java objects and thus cannot support platform- and language-independent communication. Nevertheless, we ignore this fact in the further discussion since we are concerned with communication on a higher level.

³In the rest of this paper we call an agent environment a *place*. This is the terminology used in AMETAS.

This question has at *least two* answers. The first one considers the agent paradigm as such. The discussion about agenthood has not stopped and probably never will. Different people assign different properties to agents and make up their own notion of agenthood. However, one of the few concepts that can be found in nearly every such definition is *autonomy*. Agents are said to be autonomous when they act without the constant need to be directed, monitored or controlled by any other entity. They can take their own decisions to fulfill their goals. We argue that method calls, local or remote, severely limit the autonomy especially of *mobile* agents and thus violate the basic properties of agenthood. Therefore, one could say that agents that communicate via method calls are no agents!

When agent A calls a method on agent B, one part of B's code runs under foreign control. When it is a local method call, agent A directly controls the method's code within its own flow of control⁴. In the case of a remote call, the runtime environment controls the execution of B's code. This in itself already presents a violation of B's autonomy since A decides when and how to change B's internal state. We assume that generally the method call will alter B's state according to the parameters submitted. An even bigger problem is given when B decides to migrate before the method execution has completed. Clearly the code that is executed by A and the state on which that code works have to stay until the call is completed. In RMI and CORBA a server has no means to interrupt a call to one of its methods as will be shown below. B can not migrate! Moreover, it might happen that by specifying certain parameter values, A implicitly decides how long B will not be able to migrate. B's autonomy is temporarily lost. Developing the picture further on, we get to the second answer to the above question: the technical aspects. What happens if B ignores the fact that A is currently executing B's method and B migrates? In this case, B is effectively cloned since the runtime environment holds a reference to B at least until the current call is completed. So besides the new instance created through the migration, B continues to exist on the host where A called its method. Inconsistencies will be unavoidable. Some considerable problems arise from the fact that RMI, for example, simply was not designed to handle *mobile servers*⁵. A RMI server, which is what an agent will be if it lets other agents call its methods over RMI, has no possibility to instantly remove the reference held by the RMI runtime environment. RMI servers were never meant to be mobile and therefore do not provide this vital function. In JDK 1.2 this was changed by introducing the `unexportObject` method. The server can remove its reference from the RMI runtime system even if there are pending calls. These calls will result in an exception. But still B cannot do anything about a method call in progress. We assume that explicit conditional return statements within B's method are not an option. By putting this burden on the agent programmer, any advantage introduced by using RPC-like communication would be nullified since he would have to know about the internals, the dangers and the little tricks to get around them. CORBA supports the function `deactivate_object` which essentially has the same semantics as `unexportObject` in RMI. The current method call will execute to completion.

Coming back to Java and RMI, the only way to prevent B from migrating during a call to one of its methods is to synchronize all exported methods and the `go` method on the agent object providing the `go` method. Method synchronization is the only option here, since synchronized blocks still leave the invocation and some variable initializa-

⁴We do not consider local method calls in the rest of the paper since the necessity of holding an object reference of the callee has major security implications. Therefore, this practice is prohibitive.

⁵The term *mobile server* is used to indicate that RMI-enabled mobile agents are treated as normal servers by the RMI runtime.

tions unsynchronized. Even if the programmer starts the synchronized block right after the method signature, the compiler will optimize the byte code by putting statements that need no synchronization before the block. According to Murphie's Law this always leaves enough time for the agent to execute go after one of its methods was remotely called and before the synchronized block was entered. Unintentional and in most cases unrecognized cloning is the result. The necessity to use method synchronization has severe consequences for the design of an agent. You cannot synchronize methods of two different objects with each other. Besides, synchronization is well-known to be a major performance killer and results in deadlock-prone software. For these reasons synchronization should be used with extreme care. Another problem arises from the fact that, in the above example, A is also blocked by the method call until it returns or a socket timeout occurs.

The statements given above show the obvious mismatch between mobile agents and RPC-like communication. Other technical difficulties that are not detailed here include, for example, the localization of a mobile agent to be able to call its methods and achieving the atomicity of localization and communication to prevent the target agent from disappearing in between. These problems are still unsolved but present major issues when using synchronous communication between mobile agents. Summarizing the above statements:

If we consider autonomy as one of the major properties of agenthood, then objects calling methods on each other are no agents. There is no simple way to work around this. And even if there was, the technical problems and limitations involved are a high price to pay.

The Human Factor

Most agent researchers limit their view to existing mechanisms for distributed communication. We believe that inter-agent communication should be inspired by the interaction schemes used by real-world autonomous, mobile entities like human beings. The means for human communication are essentially email, fax, telephone, mail and, of course, face-to-face conversation. None of these is actually synchronous. At first, we may think that telephone calls are synchronous, but answering machines and voice boxes render it asynchronous. We can choose if, when and even where to pick up the phone. If we do not, the caller can leave a message. Email as one of the most heavily used communication mechanisms is so popular because of the very fact that it is fast and asynchronous. In a face-to-face conversation you don't grab your dialog partner's brain to extract the information you need. Instead, you ask him to actively give the desired information to you. He may give it to you or he may reject your request. All of the mentioned communication means use some kind of *self-contained messages* that are deposited in some kind of *mailbox* for asynchronous processing. Nobody would consider the abstraction of a remote method call appropriate for inter-human communication. It would be considered an abuse if somebody else simply uses your abilities for some unknown period of time limiting your freedom. Why should IT-world autonomous, intelligent beings tolerate this?

The AMETAS View on Agent Communication

In the previous sections we saw that RPC-like communication is likely to cause problems when applied for inter-agent communication. But how does AMETAS deal with this problem?

In AMETAS, agents communicate by using asynchronous messages, hence the name. The messaging system is the one and only communication mechanism in AMETAS. Therefore, we do not run into the problems mentioned above. Agents deposit messages for each other on an agent place and other agents retrieve these messages actively. Below we will see that this does not necessarily mean that polling has to be applied. Since every agent is responsible for actively retrieving its messages, there is no blocking, no constraint to stay and passively wait until some method has finished executing and no technical problems involved with merging incompatible paradigms. But since there is no such thing as a free lunch, the resulting autonomy comes at a price:

- The foremost limitation that agent programmers might feel is the new and unfamiliar view on communication. After all, sending messages manually is not as convenient as calling a method.
- Secondly, distribution is explicitly dealt with. While in an RMI-based system an agent might miraculously acquire a reference of another remote agent without having to care about its whereabouts, AMETAS agents are location-aware. That is, they rely on the fact that an agent knows where it is and where it has to go when it wants to receive certain messages.

Both aspects can be justified by stating that unlike conventional distributed objects, agents need awareness, not transparency. An agent has to know where it is and where it has to go in order to fulfill its task. Location matters to it and having control over its communication scheme might be crucial in cooperative multi-agent applications.

The AMETAS Messaging System

Every AMETAS place offers a core service called Post Office (PO) which takes messages from a sender and stores them until an eligible receiver actively retrieves them (Figure 1). The agent that deposits a message is called the *sender*, while an agent that queries the PO for messages addressed to it is called the *inquirer*. If any messages arrived for the inquirer, it becomes the *receiver* of these messages. An addressing scheme is employed which allows the sender to specify the set of eligible receivers based on agent names and wildcards. An agent name, also called *Place User ID*, consists of several fields, each of which can be ignored during the matching of receiver and inquirer ID by using wildcards in the receiver ID field of the message. This is a simple mechanism for achieving unicast as well as multicast and broadcast messages.

The PO dynamically holds a set of mailboxes. A mailbox is not bound to one individual agent (i.e. to a single receiver) but rather to a receiver address. This might not seem to make a difference at first glance, but considering wildcard multicast addresses, it does make a big difference. As stated above, a receiver address can incorporate a number of different receivers when wildcards are used. If we would maintain a mailbox for every individual agent, a problem arises: Any non-unicast message would have to be cloned and the clones would have to be distributed over the individual mailboxes upon deposition. This can be an extremely expensive procedure, considering execution

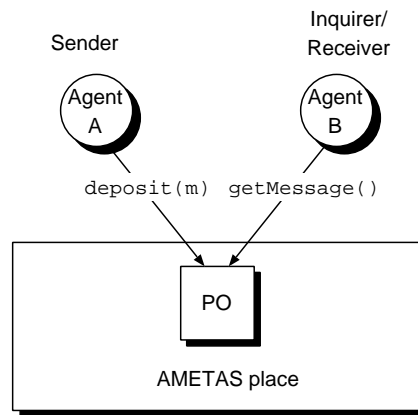


Figure 1: Communication via the PO

time and memory usage. With our mailbox system, each message has to be stored only once, which is very efficient. The downside is that every mailbox has to be checked when a potential receiver requests its messages. But since this check is an inexpensive match between the inquirers name and the addresses associated with the mailboxes, the overhead is small. Each message has a time-to-live field which is used to determine which messages are too old and have expired. Periodic cleanups ensure that expired messages and unused mailboxes are garbage collected. Therefore, the PO can be kept small which reduces memory consumption.

Higher-Level Communication Mechanisms

The messaging system is simple and small but efficient and can be used either directly or to implement more complex and powerful communication mechanisms like the ones introduced in this section. The main purpose of these mechanisms is to claw back some of the convenience and abstraction lost by introducing asynchronous messages as the one and only means for communication.

Using Events to Deliver Messages A major step towards abstraction is taken by using the AMETAS event mechanism to deliver messages. One drawback of a pure messaging system is polling: Agents have to actively fetch the messages addressed to them, often querying the PO in vain. This wastes system resources and is considered a *DON'T* in computer science in general. Polling can be easily avoided by using one of the other AMETAS core services: the event system. Events are generated by the AMETAS core when certain situations occur that agents should know about. One such situation is the arrival of a message in the PO. Every time a message is deposited, an event is fired by the PO which is processed and forwarded to registered agents by the *Event Manager*. Agents can register for specific events and explicitly specify which events they wish to receive. In the case of message events this means that upon registration, the agent supplies filter objects that specify the senders and the receivers⁶ of the messages it is interested in.

⁶Keep in mind that the *receiver* specified in the message header can match a number of agents using wildcards.

If an agent has registered for certain messages and a matching message arrives, it is immediately notified through a call to one of its methods. Note that this is a dedicated method which is called only by the local AMETAS core, not by another agent. It lies in the responsibility of the agent programmer to process events as quickly as possible to avoid blocking. The message delivery using events is depicted in Figure 2.

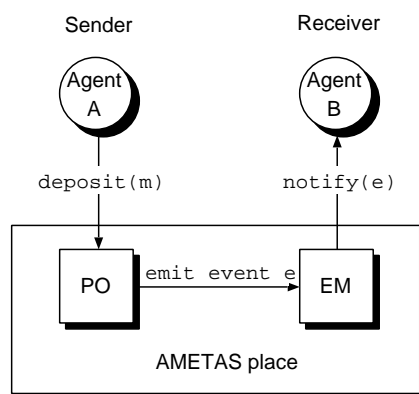


Figure 2: Using events for message delivery

The usage of message events eliminates the need of polling and presents an effective mechanism to deliver messages in a timely fashion. Besides, message events behave more like normal method calls in that there is now an active and a passive communication partner. This paves a way for the high-level *Service Proxy* mechanism introduced next.

Service Proxies as a Means of Abstraction Even though services communicate in just the same way as agents do in AMETAS, they are not called agents. The reason is that services do not exhibit autonomous behaviour, are non-mobile and may access system resources. The typical *wait-and-serve* interaction scheme applied by services contradicts agenthood. The same is true for user adapters.

This non-agenthood makes services eligible for remote method calls. Therefore, we build a concept called *Service Proxy*⁷ on top of our messaging system to achieve the abstraction of a method call with the caller being any Place User and the callee being a service. A Service Proxy is an object provided by a service programmer that encapsulates the process of building (marshaling) and sending a message, waiting for a result, enforcing timeouts and unmarshaling the received response. It could be compared with a stub object in RMI or CORBA with the difference that the underlying protocol is not IIOP or the RMI transport protocol, but AMETAS messages. For each *service method*, the Service Proxy contains a real method that takes care of the details of the service interaction and communicates with the service taking parameters and returning result objects. An agent that wants to use the service instantiates a proxy object and calls the respective methods for service interaction. The caller is blocked until a result is returned or a timeout occurs. The agents has the illusion that the service is within its own object, i.e. local to it.

Service Proxies are never used for communication between two agents. They are specialized for service requests. For the reasons mentioned above, services have no

⁷Service Proxies are scheduled to be a part of a future AMETAS release.

autonomy to lose. Remember that the callee was our major reason of concern when using method calls between agents since the callee loses its autonomy. Agents (or rather agent programmers) choose to use proxies because especially when interacting with a service, a blocking method call is most often what they want. In most cases we need the result to carry on working. It is important to note that to the outside world the appearance of the caller does not change. After all, the proxy is a part of the agents own state and viewed from the outside, the agent still sends and receives AMETAS messages. The new abstraction is limited to the internal structure of the agent.

Remote Communication A straightforward extension to the AMETAS messaging system are *remote messages*. Instead of depositing a message to the local PO, an agent may deposit it to the PO of another place. Additional information is included in the message header identifying the source place. The message is transferred to the remote place and delivered to its PO. On the remote place the message can be fetched or delivered via an event just like a locally deposited message.

Messenger Agents One drawback of remote messages is the lack of implicit security. Since the PO ensures the secure storage of messages and since there is no dangerous transmission involved with depositing messages locally, they are not encrypted or signed. Sending a remote message involves a transmission upon which an attacker could intercept the message, read it, manipulate it or discard it. Since remote messages are a very simple enhancement of the messaging system, no additional security mechanisms are introduced. Therefore, remote messages should only be used in secure environments.

Secure remote communication may be achieved by exploiting the implicit security mechanisms used for transmitting agents. An additional layer may be introduced between the messaging system and the application. This layer would consist of special mobile agents called *messengers* which transport messages from one place to another and can engage in interactions that go far beyond a simple message deposition. Whole conversations could be led locally by such a messenger on behalf of the remote agent that sent it. Depending on the intelligence and *eloquence* displayed by a messenger, it would be a light-weight communication assistant. Due to the code caching mechanism used in the AMETAS migration subsystem, the code of a messenger will be transmitted to every place only once. On every subsequent migration, only the state is transferred which eases the expected network load considerably.