

AMETAS

Good Migrations.

AMETAS White Paper Series

by Klaus Herrmann
and Michael Zapf

On Writing Agents

July 2000

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

www.ametas.de

Writing Agents

The mobile agent paradigm is radically different from any other programming paradigm known so far. Active code mobility requires the programmer to learn some new ways to design his code. If we start to write code without considering the little subtleties introduced by migration for example, we might end up with applications that do not behave as expected. Other issues concern the new ways of communicating and the general structure of agents. In this White Paper we try to give an overview on writing AMETAS agents to give the reader a feeling for this new way of designing software. However, we would suggest you to read the AMETAS tutorial available at www.ametas.de before you start designing agents.

Weak Migration – Some Notes

The biggest difference between normal objects and mobile agents is, of course, that mobile agents can actively migrate. AMETAS adopts a *weak migration* scheme. That is, an agent may call the `go` method to transfer itself to another place. At the destination place the agent does not continue executing the line after the `go` statement. As opposed to *strong migration*, where this is actually done, AMETAS agents are restarted by calling the `invoke` method each time they migrate. At first glance, this seems to be rather inconvenient and one might ask why strong migration is not used. There are two good reasons for that:

1. In Java, strong migration can only be achieved by modifying the virtual machine. The agent system has to be able to get the execution stack from the VM and transfer it with the agent's code to the target place. Modifying the VM renders it platform-dependent and is a very burdensome task. Other techniques require an explicit checkpointing at the application level to coarsely determine the execution stack.
2. Transparent migration, as strong migration is also called, is not appropriate for mobile agents. Changing into a formerly unknown environment necessitates a re-orientation to see what resources and which other Place Users are present. An agent cannot blindly resume execution and ignore the new circumstances it has been thrown into. Therefore, a restart is implicitly necessary for the agent to be able to reinitialize itself.

Now that we have seen the necessity for a restart after each migration, how does this influence our code?

Designing the `invoke` Method The `invoke` method is called after every migration. Its structure can usually be split into three parts. The first one is the part responsible for the first initialization which is only executed once. A boolean member of the agent indicates whether it is started for the very first time or whether it just recovered from a migration. The second part is executed on every restart and is responsible for adapting to the new environment. Here the agent might try to get in contact with certain services and register itself as Event Listener. The third part is doing the actual work. It should be executed after the initialization if the agent was able to get every resource it needs to fulfill its task. Actually, AMETAS does not predefine this structure or direct the programmer to follow it. The programmer starts out with an empty `invoke` method and has to implement the structure by himself since it is only a guideline.

Migration Errors Sometimes a migration cannot be executed due to some unexpected conditions. Maybe the agent was rejected by the target place because it does not comply with the security requirements. In this case exceptions are thrown by the `go` method that indicate the reason for the migration failure and must be caught and properly handled. This is the only case in which the execution continues after the `go` statement. So anything below a `go` is error handling code.

Other Fatal Errors If an agent causes a fatal error that is not caught and handled within the agent's own code, the agent would normally be terminated abruptly. AMETAS deals with this by catching any uncaught errors and exceptions and calling the agent's `recoverError` method to inform it about the error. At this point, the `invoke` method is terminated and the agent is given a last chance to deal with the error. It could go on working within the `recoverError` method but another uncaught error would cause its immediate termination. So the `recoverError` method should be viewed as an emergency exit rather than a way to continue work.

Local Variables Since every migration stops the agent, transports it and restarts it, all local variables present within the method scope at the time of invoking the `go` method are lost. Information that is considered valuable has to be stored in instance variables of the agent object since these define the agent's state which is transferred.

Transporting Important Information Not all the information that is stored in the agent's state is really transported to the next host. The `transient` modifier of the Java language becomes quite important in this respect. Its purpose is to indicate whether an instance variable of a class is to be serialized upon object serialization. A transient agent member will be discarded on every migration. On restart, such variables are automatically initialized in a way that is defined by their type. Any non-transient variable (variable without the `transient` modifier) is actually serialized, transported and recreated at the target place. The `transient` modifier should be used to keep the agent's state as small as possible by using it on variables that are reinitialized after a migration anyway.

Migration from inside a Loop A code fragment like the one given in Figure 1 will not work as we expect it to work. The reason is that the call to the `go` method implicitly terminates the agent and reinitializes all local variables including the loop counter. The result in this special case will be repeated migrations to the first place.

It should be noted though that migration from inside a loop is the only real subtlety that could catch out programmers. The other implications are quite straight forward and easily recognizable.

Agent Size An issue, not only in weak migration but in any form of code migration, is the size of the transported agent's code and state. This size should be kept to a minimum to minimize bandwidth usage. On the other hand, programmers want powerful and well-structured agents which usually results in more data and code being transmitted. This is an ever-present trade-off that has to be reconsidered every time we start programming a new agent. A sensible compromise has to be found. In normal Java programming this is normally the least of all worries the programmer has, but in agent

```

private String[] m_asPlaces =
    {"firstplace.mydomain.de",
     "secondplace.mydomain.de", ...};
...
public void invoke() {
    ...
    for( int i=0; i<m_asPlaces.length; i++ ) {
        // do some work
        ...
        m_Driver.go(m_asPlaces[i]);
    }
    ...
}

```

Figure 1: Migration inside a loop

programming it should be one of his primary worries. Experience has shown that normally the code is several times as large as the agent's state. So writing small code is important.

The Driver

The only way in which an agent may interact with its environment is by calling methods on its *driver*. The driver introduces a Place User specific separation for Place Users and the place. Important methods like the `go` method are present and may be called in the driver (see Figure 1). The driver may add some information to the call and forward it to the place. For example, the agent's ID is necessary for using most core services to identify the caller (the agent) and maybe decide whether it is eligible to execute this call. This ID is always added by the driver for security reasons. Thus, no Place User can accidentally or intentionally submit a wrong Place User ID. The driver can be seen as one of AMETAS' most important mechanisms to enforce a strict separation and security measures.

Communication via Asynchronous Messages

As the White Paper on communication states, Place Users can only communicate by depositing messages for each other. A typical example of depositing a message can be seen in Figure 2.

```

Object[] aobjBody = {"HELLO_MESSAGE",
                    "Hello world!",
                    new Integer(42)};
m_Driver.depositMessage(puidReceiver, aobjBody);

```

Figure 2: Depositing a message

We see that depositing a message involves the creation of an object array as message

body and a call to the driver method `depositMessage`. Arguments to that call are the ID of the receiver and the messages body. The receivers ID may be acquired by a request to the place's mediator. The receiver may simply call the driver's `getMessages` method to retrieve all messages deposited for it. There are some other driver methods which take different parameters to deposit a message. These can be used for instance to provide all the message's header fields explicitly, which are set to useful default values in the example. Another set of methods are specifically tailored to service requests. Such requests are also done by simply depositing a message but most clients want to be blocked until a response arrives. This blocking is achieved by separate methods within the driver.

Starting and Initializing an Agent

Agents are mostly started by other Place Users, especially by user adapters. The question arises how an agent can be initialized with concrete parameters. Such a parameter could be an itinerary telling the agent which places to visit. These parameters also have to be given to the agent via a message. The driver essentially offers two different methods for starting a new agent: `requestPUStartup` and `spawnAgent`. The first one loads a Place User (agent, service or user adapter) from the local file system and starts it. The second method is used to start an agent that is present in the calling Place User's object. For example, one agent can contain another (inactive) agent when it travels through a network. It can decide when and where the contained agent must be started. If the time has come and the place is right, it calls `spawnAgent` and the second agent's code is read from its own code repository and is started. Coming back to the initialization question, each of the two methods takes a class name and a message which is deposited for the new Place User before it is started. One of the first actions taken by the newly created Place User is to query the Post Office for that initialization message and process it. This has to be done explicitly in the Place User's code.

Event Handling

Using the event system to get notified when a specific message is deposited is a way to avoid polling. This is explicitly described in the White Papers on communication and events. To participate in the event system, an agent needs to implement the `AMETASNotifiable` Interface. This interface provides a single method (`notifyListener`) that is called upon the generation of an event the agent has registered for. In our case this would be message events which contain a message that was deposited for the event's receiver. Figure 3 shows what event handling code looks like.

First we register the agent as an Event Listener listening for received messages. This is done in the `invoke` method upon initialization. The object of class `AMETASMessageMask` is a filter that specifies which messages are interesting to us. The `null` arguments are interpreted as wildcards and using three of them, as in the example, indicates to the Event Manager that we are interested in every message that is deposited for the agent. After the registration, the agent will get notified upon any message that is deposited for it via a call to the `notifyListener` method. In this method it extracts the messages from the received event and processes them in a loop. Using the extended event handling mechanism provided by AMETAS, this can even be done in a more convenient and flexible way.

```

public class MyAgent implements AMETASNotifiable {
...
    public void invoke() {
        ...
        // register for message events
        Object aobjArgs[] =
            {new AMETASMessageMask(null, null, null)};
        try {
            m_Driver.registerEventListener(
                AMETASMessageEvent.MESSAGE_RECEIVED_EVENT,
                0, aobjArgs);
        }
        catch(EventException ex) {...}
        catch(UnauthorizedEventException uex) {...}
        ...
    }
...
    public void notifyListener(AMETASEvent evt){
        AMETASMessage ames[] =
            (((AMETASMessageEvent)evt).getMessages());
        for(int i = 0; i < ames.length; i++){
            // process the messages here
        }
    }
}

```

Figure 3: Handling message events

Putting it All Together

In the preceding sections we discussed some issues that are important to the developer of AMETAS agents. Let's now give a brief summary of the things that a programmer has to take care of. We try to order this list beginning with the decisions that should be taken first:

1. **Power versus migration frequency:** The decision about how powerful the agent must be and at the same time how frequently it has to migrate is usually a very fundamental one. It is of course highly influenced by the basic task the agent has to fulfill. This decision will have an effect on the code size of the agent. It is vitally important to achieve a good balance between power and size.
2. **Extended or simple event mechanism:** The answer to this question depends on the amount of different events that have to be processed. Since in most cases the majority of events will be message events, it depends on how frequently the agent is communicating. The simple mechanism is used when the overall amount of events can be handled by relatively small code fragments. The extended mechanism should be used when a single handler method becomes very large and difficult to structure. A second reason could be the necessity to have several independent event handlers not mixed into one method.

3. **Structuring the agent's working phases** is done by deciding when and where and how to evaluate initialization messages, to execute post-migration initialization and to do the main tasks. Traditional design techniques for structuring software are used to do this.
4. **Handling errors** is especially important for mobile agents since it will not be apparent to the user if one of his agents ceases to work on a distant place. It is typical that large portions of an agent's code will be dedicated to handling exceptions. We give the advice to catch every single exception that can be thrown by the code individually and at least write a short log message so that information is available on the error. As a safety precaution it is also advisable to include a `catch(Throwable)` around the `invoke` method to ensure that no error can terminate the agent. If this is not done, implement the `recoverError` method to handle terminal errors.
5. **Transient variables** can be defined in the finishing stages to reduce network load when migrating. Of course the important instance variables that have to be transported to every node should be identified early in the design process.