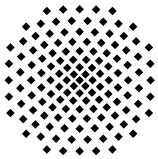

Universität Stuttgart
Fakultät Informatik

A Protocol for Orphan Detection and Termination in Mobile Agent Systems

Joachim Baumann

Email: Joachim.Baumann@informatik.uni-stuttgart.de

Institut für Parallele und Verteilte
Höchstleistungsrechner (IPVR)
Fakultät Informatik
Universität Stuttgart
Breitwiesenstr. 20 - 22
D-70565 Stuttgart

A Protocol for Orphan Detection and Termination in Mobile Agent Systems

Joachim Baumann

Bericht 1997/09
July 1997

A Protocol for Orphan Detection and Termination in Mobile Agent Systems

Joachim Baumann

IPVR (Institute for Parallel and Distributed High-Performance Systems)
Breitwiesenstraße 20-22
D-70565 Stuttgart

Phone: +49 711 7816 218
Fax:+49 711 7816 424
EMail: Joachim.Baumann@informatik.uni-stuttgart.de

Abstract

Orphan detection and termination in distributed systems is a well researched field for which many solutions exist. These solutions exploit well defined parent-child relationships given in distributed systems. But they are not applicable in mobile agent systems, since no similar natural relationship between agents exist. Thus new protocols have to be developed. In this paper one such protocol for orphan detection and agent termination is presented.

First we present two approaches, the energy concept and the path concept. The energy concept is a passive termination protocol, and the path concept is a protocol for finding agents, that can be used to implement active termination.

The ‘energy’ approach is based on the idea that an agent is provided with a limited amount of energy, which can be spent in exchange for the resources used by the agent. From time to time the agent has to request additional energy from its creator. The agent is terminated as soon as the energy falls to 0. This approach to agent termination implicitly implements orphan detection, i.e. if the creator has terminated, the dependent agents are killed as soon as they have no energy left.

The ‘path’ approach uses a chain of proxies. As soon as an agent leaves a location, a proxy is created that points to the new location. By following the chain of proxies, the path, one can find any agent, and consequently terminate it.

Both approaches have disadvantages. By merging them on different levels we create a protocol that combines the advantages of both approaches, and at the same time minimizes the disadvantages.

The ‘shadow’ approach uses the idea of a placeholder (shadow) which is assigned by the agent system to each new agent. The shadow records the location of all dependent agents. Removing the root shadow implies that all dependent shadows and agents are terminated recursively. We demonstrate that the shadow approach can be used for termination of groups of agents even if the exact location of each single agent is not known. We show that shadow protocol is less fault-sensitive than the path approach and needs less communication cost than the energy approach.

1 Introduction

A mobile agent is seen as a piece of software roaming the network on behalf of a user, e.g. searching for information in different databases, buying a flight ticket and renting a car, or trying to find the cheapest flower shop. Mobile agents seem to be the solution to many of the problems in the area of distributed systems. But while the idea of mobile agents is quite appealing, and while many researchers are working in this area, some very important problems have not been solved. Most of the research concentrates on providing the basic system support for migration, communication, the security of the platform underlying the agent system and for the asynchronous operation of agents. Some solutions for these problems already exist and have been implemented in different agent systems (e.g. [StBaHo96], [Aglets97], [White94a], [GenMag97, [BaTsVi96]).

But in the area of termination and orphan detection in agent systems no solutions have been implemented until now.

Termination and orphan detection in an agent system are very important both from the user's and from the system side, because a running agent uses resources which are valuable to both user and system. The user has to pay for resources, and the system has only a limited amount of them. So if the user does not need the results of a distributed computation in progress (e.g. a group of ten agents on different systems), he wants to be able to terminate it to be able minimize the resulting cost. Orphan detection guarantees that even if the termination mechanism has failed, the now useless agents can be determined by the system and ended, thus freeing the resources they have bound.

In this paper we will investigate two approaches, the 'energy' approach and the 'path' approach. After discussing the advantages and disadvantages of both we present a new protocol, the so-called shadow protocol, that combines both approaches.

In section 2 we define the meaning of orphan detection and termination in Mobile-Agent-Systems. Section 3 presents our agent model, and section 4 defines the requirements for protocols in the area of Mobile-Agent-Systems from different points of view. In section 5 we discuss the energy approach, followed by the path concept in section 6. In section 7 the shadow protocol is presented with different extensions and optimizations. Section 8 compares the different protocols presented in the light of the requirements found in section 4, section 9 presents related work, and in section 10 the conclusion and outlook is given. The appendix A contains the different protocols expressed in pseudo object-oriented notation.

2 Orphan Detection and Termination in Mobile-Agent-Systems

In this section we will present our view of orphan detection and termination in Mobile Agent Systems, which differs substantially from the notion taken in the area of distributed systems.

2.1 Orphan Detection

Orphan detection is a means to find out if an agent is still needed by the application using it. If it is not, it is an orphan and can be terminated. This decision is by no means trivial because normal parent-child relations as in distributed systems, e.g. Client-Server relationship or in the RPC-paradigm, the relationship between caller and called, might not exist. Thus to provide such a mechanism we have to artificially establish a structure that provides some sort of relation.

Typical relationships in “normal” distributed systems are:

- the Client/Server relationship, in which a server activity has a clear relation to a request (or number of requests) sent by a client. The server can declare its activity as orphan as soon as it gets to know the client exists no longer (see e.g. [Tanenb95, ComSte93]).
- distributed systems with migrating processes, where a process has at every time a parent process (the creating process) it depends on (see e.g. the V distributed system [Goscin91). This allows to determine if a process is an orphan by examining the parent process.
- garbage collection in distributed systems, where some distinguished objects, called root objects, are known to be no orphans (e.g. because they communicate with a user). An object having neither a reference (direct or indirect) to such an object nor the ability to get one such reference in the future is declared an orphan and terminated (see e.g. the work on Stub Scion Pair Chains [Shapiro92]). While this is not intuitively seen as being in the same problem area as the first two problems, it is e.g. presented in [BagPiu96], how these can be used to track resource allocation in mobile environments.

The main problem with orphan detection in Mobile Agent Systems is that in contrast to “normal” distributed systems, there does not necessarily exist an easily identifiable relationship between the participants in a distributed computation. Let us assume an agent creates another agent, informs it to report the result of its task to a third agent that not even exists yet, and terminates. The second agent now depends on the third agent, not on the one that created it. But this third agent might only exist at some time in the future to receive the result. Is now the second agent an orphan? No, but a system cannot determine that without additional information.

In each of the three areas mentioned above the dependency between the participants defines, in a natural way, what an orphan is and how it can be detected. This is not possible in an agent system, because no dependency is defined by an underlying paradigm, as in Client/Server interactions, with migrating processes, or in the area of distributed garbage collection.

What has to be done is to create such a relationship artificially. This of course redefines the term “orphan” in the context of this dependency. This in turn restricts the area of use for a protocol exploiting this dependency to the only those problem areas in which this artificial relationship doesn’t collide with the problem solution. It follows that if such a road is taken there cannot exist the one solution to orphan detection for all possible relationships. There has to be a suite of protocols tailored to different needs instead. Relationships might be dictated by the interaction patterns of agents, by application specific needs or simply by the preference of the programmer. We will discuss three possible means to define such relationships in the course of the paper.

There is one problem with such an artificial construct. It cannot be deduced from the environment, i.e. not locally by examining only the agent and its communication pattern, if it is an orphan. To decide this, additional information has to be conveyed, normally by the means of communication, which implies additional cost for the protocol. One concept that gets away without an explicit structure and thus without the additional communication cost (on the system level) is the energy concept that employs the use of resources as the determining factor.

2.2 Termination

In this paper we view the termination problem in the area of agent systems as the problem of how to terminate i.e., to end an agent that is somewhere in the network. This is different from

the definition in the area of distributed systems, where the termination problem is the problem how to determine when a distributed computation has been finished [Matter89].

Termination in the context of mobile agents can be done two different ways. The first is to somehow find the agent in the network and terminate it then (active termination).The second is to change the status of the agent in question to that of an orphan, and let the system remove it (passive termination), which does not necessarily imply searching the agent.

3 The Agent Model

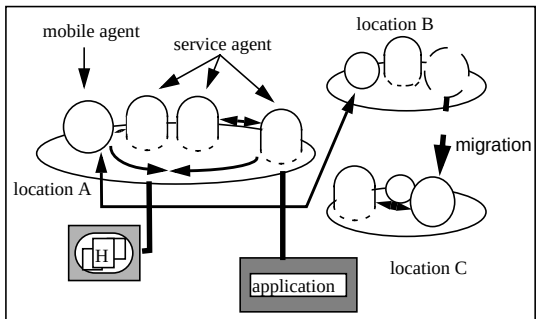

Figure 1: The Agent Model

In this section we will give you only a short overview of our agent model, that has been described in much more detail in [StBaHo96] and [BaumEA97]. Our model of an agent-based system - as various other models - is mainly based on the concepts of agents and locations. An agent system consists of a number of (abstract) locations, being the home of various services. Agents are active entities, which may move from location to location to meet other agents and access the locations' services. In our model, agents may be multi-threaded entities, whose state and code is transferred to the new location when agent migration takes place. Locations provide the environment for safely executing local as well as visiting agents.

Each agent is identified by a globally unique agent identifier. An agent's identifier is generated by the system at agent creation time. The creating location can be derived from this name. It is independent of the agent's current location, i.e. it does not change when the agent moves to a new location. In other words, the applied identifier scheme provides location transparency.

A location is entirely located at a single node of the underlying network, but multiple locations may be implemented on a given node. For example, a node may provide a number of locations, each one assigned to a certain agent community, allowing access to a certain set of services or implementing a certain prizing policy. Locations are divided into two types, depending on the connectivity of the underlying system. If a system is connected to the network all the time (barring network failures and system crashes), a location on this systems is called *connected*. If a system is only part-time connected to the network, e.g. a user's PDA (Personal Digital Assistant), the location is called *associated*.

4 Requirements

In this section we define the requirements for orphan detection and termination protocols in Mobile Agent Systems. One set of requirements can be derived from the general requirements found in termination and orphan detection protocol in the area of distributed systems (see e.g. [Matter89]). These describe properties of a protocol for distributed systems in general:

- No restriction concerning the communication model.
- No restriction concerning the order of messages.
- No acknowledgement of every single message.
- No restriction concerning the topology of the agent system.
- No global time.

- No freezing of the application or the agent system to obtain global knowledge.
- Well-defined fault semantics (this includes partitioning of the network, crashes of agent or system).

Specific requirements for an orphan detection and termination protocol can be obtained by examining the protocol from two viewpoints, the system's and the agent application's. While the system requirements define mainly security-related needs, the application requirements are primarily concerned with the suitability for real applications.

One requirement can be derived from both the system's and the application's point of view:

- The impact on the communication costs should be as low as possible.

The system's point of view gives us the following additional requirements:

- An agent should have only a limited life span.
- The algorithm should have no loopholes. This should hold true for malicious agents on one hand, and for faulty agents on the other hand.

The following requirements can be derived from the application's viewpoint. These properties enable a programmer to use an orphan detection and termination protocol

- An agent should be able to create other agents.
- The system checks if an agent is an orphan (not the agent itself).
- It must be possible to prolong the life span of an agent.
- The concept should support both active and passive termination.

5 The Energy Concept

In this section we present the energy concept, and discuss its advantages and disadvantages.

5.1 The Idea

In its life an agent needs resources, cpu time, memory, I/O, and uses services provided by the system. Let us assume that each service and each resource consumes some of the energy of the agent. At the beginning of its life the agent gets an amount of energy, and if all its energy is consumed, it is defined an orphan and can be terminated.

This provides a simple means of determining orphans with the advantage that the activity (i.e. the use of resources) of agents determines their life span. The disadvantage of this simple approach is that if a problem needs more energy to be solved than the application assumed, the agent is not able to finish its task. For instance parallel searches in hierarchical organized data can be problematic. In such cases it would be very helpful if the agent (or agents) could request additional energy from the application. An agent that detects that its energy runs low could send a message to its application and go to sleep, i.e. consume only small quantities of energy. It does not matter if the application reacts at once, as long as it is guaranteed that in the remaining life span of the agent the message arrives. This way, the concept is resilient against short-timed network faults and system or application crashes. The maximum time for such faults is simply the time the agent can live with the remaining energy. The application now decides if the agent gets more energy. If the application decides to serve the request, it sends the additional amount of energy to the now known location of the agent.

It has to be ensured though that there is no possibility of outmanoeuvring the mechanism. If an agent creates another agent, this agent cannot get the same amount the creating agent originally had. Otherwise a malicious agent can create a child agent that sleeps until the original agent dies, and then in turn creates a child, thus gaining infinite energy. The obvious solution for this problem is that the energy the new agent gets has to come from the creating agent. Another possible loophole exists, if the maximum amount of energy an agent can hold is unlimited.

5.2 The Protocol

The protocol as a whole is presented in appendix A.1. Here we will discuss the different parts separately. The protocol is presented in an object-oriented pseudo notation.

The location on which an agent lives, has to decrease the energy of the agent in regular intervals to guarantee the finite lifetime. The decrease is dependent on the activity of the agent (the agent can be put to sleep or awakened by system methods). If the decrease results in the agent having no energy left, the system terminates the agent (see Figure 2).

Every system method called by an agent, every service requested has to check if the agent has enough energy remaining. If this is the case, the energy amount is taken from the agent and the system method called. Afterwards a check for the remaining energy has to be made by the system. This check examines the remaining energy and if it is under a threshold determined by the agent programmer or the agent user, the agent is signalled that the energy is low. Additionally a flag is set that the agent is in its refilling stage. This prevents that the agent is signalled more than once (see Figure 3).

The agent itself decides how to react to this signal. One of the possible reactions is shown in Figure 4. Here the agent send a message to its home asking for more energy, and goes to sleep.

The home system receiving the message can react, e.g. by asking the application if the agent should continue its work. If an amount of energy is sent, the receiving agent is informed, its energy is added to and the agent is wakened.

```
Regular Intervals:
    for each agent
        agent.energy = agent.energy - agent.livingCost();
        if (livingAgents.find(agent) != null)
            checkLowEnergy(agent);
        if(agent.energy == 0) remove agent;
sleep(agent)
    livingAgents.remove(agent);
    agent.livingCost = location.sleepCost();
wake(agent)
    livingAgents.add(agent);
    agent.livingCost = location.livingCost();
```

Figure 2: Basic Location methods

```
SystemMethod(method, agent)
    if agent.energy < method.cost
        throw NotEnoughEnergy;
    agent.energy = agent.energy - method.cost;
    method.callMethod(agent);
    checkLowEnergy(agent);

receiveEnergy(agentId, amount)
    agent = getAgent(agentId);
    agent.energy = agent.energy + amount;
    agent.receiveEnergy();
    if (agent.energy > agent.lowEnergy)
        agent.refilling = false;
        wake(agent);

receiveMessage(„energyLow“, agentId)
    [implement policy];

checkLowEnergy(agent)
    if (agent.energy < agent.lowEnergy
       AND agent.refilling == false)
        agent.energyLow();
        agent.refilling = true;
```

Figure 3: System methods

```
EnergyLow()
[implement policy]
    sendMessage(home, “energyLow”, id);
    sleep();

ReceiveEnergy(amount)
[implement policy]
```

Figure 4: Agent Methods

5.3 Discussion

The agent paradigm provides as one of its main advantages the asynchronicity of computations. By introducing the energy concept, this advantage is reduced. Using the energy concept the agents are too dependent on their applications. Furthermore, they are dependent on the application sending them additional energy in time to model asynchronous operation well. Assume an application that is started only every other week to check information gathered by its agents. This application would either have to be started more often only to send energy to its agents, or the agents would have to be able to store large amounts of energy, which would de facto circumvent the mechanism.

A second disadvantage is that the agent programmer has to implement a strategy for refilling the agent's energy. Predefined routines, e.g libraries, could be of help, but wouldn't change the problem that the programmer has the responsibility to implement the specific strategy.

The advantages on the other hand are obvious. All information to check if an agent is an orphan is available locally; no additional communication is necessary, which means that scalability is given. While the energy concept does not provide additional security for the system, it allows the identification of the agent owner. This in turn allows e.g. punishment as a reaction to the usage of malicious agents.

While the energy concept clearly has strong similarities to a currency concept, it has one major difference. If a location steals money from each agent that visits it, this stolen money can be used e.g. to buy something. If a location steals energy, the agent might need a refill of its energy sooner, but the stolen energy cannot be used. Thus no additional need for security is introduced by the energy concept.

The energy concept is by its nature an orphan detection protocol. The system can determine if an agent is an orphan simply by examining its remaining energy. Active termination cannot be done with this concept, because the agent cannot be reached by means of the protocol. The application has to wait until the agent's energy has been used up, and then can declare the agent an orphan by denying it additional energy.

6 The Path Concept

6.1 The Idea

If every agent in the agent system leaves a trail behind, then by knowing where an agent has been created, this agent can ultimately be found. This can be done by simply following the trail to its end. After finding the agent it can e.g. be terminated. This idea is not new in the area of distributed systems (e.g. Emerald [JulEA88]), but has not been used in the area of mobile agents.

6.2 The Protocol

The protocol as a whole is presented in appendix A.2. As before, we discuss the different parts separately. The path is composed by creating a proxy at the current location, storing the new location of the agent together with its id. In Figure 5 the system methods for an agent arriving and leaving a location are given. At arrival nothing has to be done, but upon leaving a new path proxy is added to the list of proxies. containing agent id and target of the migration.

```
onArrival(agent)
    agentList.add(agent);
onLeaving(agent, target)
    agentList.remove(agent);
    pathProxyList.add(newPathProxy(agent.id,target));
    SendAgent(target, agent);
```

Figure 5: Creating a trail

Finding agents can be divided in two simple steps. First we have to check if the agent is on the current location. If it is not on the current location, and no proxy exists for this agent, then either the path to the agent does not cross the current location (by asking the home location this can be avoided) or it does not exist. If a proxy exists, a request is sent to the target location stored in the proxy containing the searching location and the agent id, asking to find the agent. The receiving location determines if the agent is on that location. If that is the case it sends the information back to the original searching location. If a proxy exists, the request is sent onward. This happens until the agent is found at the end of the path (see Figure 6).

```
find(agentId)
    if (agentList.find(agentId) != null)
        return(this)
    pathProxy = pathProxyList.find(agentId);
    if(pathProxy != null)
        sendFind(pathProxy.target, this, agentId);
    else
        return(notFoundError);
receiveFind(searcher, agentId)
    if (agentList.find(agentId) != null)
        sendFound(searcher, this, agentId);
    if(pathProxyList.find(agentId) != null)
        sendFind(pathProxy.target, searcher, agentId);
    else
        sendError(searcher, notFoundError, agentId);
receiveFound(from, agentId)
    return(from);
receiveError(notFoundError, agentId)
    return(error);
```

Figure 6: Methods for finding Agents

6.3 Shortening the Path

If the information about the whereabouts of the agent are sent to the home location, the path can be shortened by setting the target of the proxy at the home location to the new location. This way the information can be updated without additional communication.

The intermediate path is now superfluous and can be removed. This is done by sending a request to shorten the path containing the new beginning of the path. A location receiving this request examines if it is the location toward which the proxy at the home location points, and if this is not the case, sends the request onward and removes the path proxy (see Figure 7). Additionally, if the agent moves back to a location it visited before, the now circular path can be cut short. This can be done using exactly the same method.

```
receiveFound(from, agentId)
    pathProxy = pathProxyList.find(agentId);
    sendMessage(pathProxy.target, shortenPath,
                agentId);
    pathProx.target = from;
    return(from);
receiveError(from, error, agentId)
    if (error == notFoundError)
        shortenPath(from, agentId);
        return(error);
shortenPath(target, agentId)
    pathProxy = pathProxyList.find(agentId);
    if(target != this AND pathProxy != null)
        sendMessage(pathProxy.target,
                    shortenPath, agentId);
        pathProxyList.remove(pathProxy);
```

Figure 7: Shortening the Path

6.4 Discussion

The path concept fully supports asynchronous operation. Agents can roam the network without having to contact their home location. In contrast to the energy concept, no additional communication cost is introduced by establishing the paths.

The disadvantages come from the path not being limited in length. The communication cost can be arbitrarily high (at most the number of locations in the agent system), directly dependent on the length of the path. Furthermore, the path depends on all visited systems being reachable. If only one of the intermediate systems is not reachable, the agent cannot be found. Thus the longer the path the worse is the fault-tolerance, and this with no sensible upper bound.

7 The Shadow Protocol

After having examined the two concepts on which the shadows are based, we present the shadow concept itself. In this section we discuss the basic Shadow Protocol with its agent proxies, the extension of making the shadows mobile, and discuss possible optimizations.

7.1 The Idea

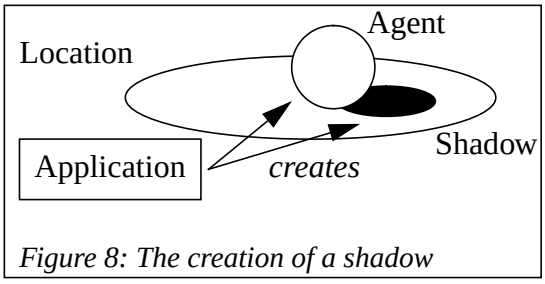

Figure 8: The creation of a shadow

In the shadow concept each application creates one or more shadows, a structure on a location that is accessible at each time. This location does not necessarily have to be the same on which the creating application runs. Each agent that is created by the application, depends on such a shadow (Figure 8). As soon as the shadow is removed by the application, the agent can be terminated by the system. Thus the agent is depending on the shadow instead of the application. As long as the shadow exists in the system, no contact of agents to the application itself or the computer system on which the application runs is necessary. In regular intervals (called *time to live*) the system checks for each agent if the associated shadow still exists. This *time to live* cannot be arbitrarily long to impede circumvention of the protocol. If the shadow does no longer exist, the agent is declared an orphan and is removed.

If an agent creates a new agent, this new agent gets the same shadow as the creating agent, and the same remaining time until the next check (Figure 9). Limiting the time span to the same of the creating agent (and not to the original *time to live*) is necessary to prevent malicious agents from living infinitely. Otherwise the mechanism could be circumvented simply by creating a new agent with again the whole *time to live* just before the life span of the old agent ends. This way the newest descendant would not be checked for an existing shadow and thus effectively removed from the control.

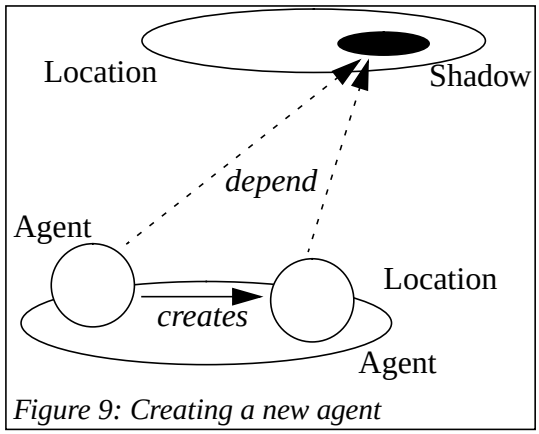

Figure 9: Creating a new agent

If a location on which a shadow resides cannot be reached, the system starts a timeout. If after the timeout the system cannot be reached, the timer is started again and the counter is decremented. If the counter is 0 and the location can still not be reached, the shadow is presumed no

longer existent and its associated agents are killed. By adjusting timeout and counter manifold reactions to communication problems can be adopted.

This protocol implements a concept akin to the energy concept, but the energy has been substituted by a *time to live*. The disadvantage of this approach is that regardless of what an agent does, it has to connect to its shadow's location in regular intervals. The advantage on the other hand is that we have an upper bound for the termination of agents through removing the shadows. This upper bound is exactly the *time to live* of the agents.

Until now the protocol only allows passive termination. By removing the shadow all depending agents are declared orphans, and after the *time to live* it is guaranteed that all agents have been removed by the orphan detection. But by adding the path concept to this protocol, we also allow active termination. This is done via the so-called agent proxies.

7.1.1 Agent Proxies

The agent proxies are structures at each location that keep track of the movement of all agents belonging to a shadow. They implement the path mentioned in section 6 that enables the active termination of agents extending the shadow's functionality. By storing the location at which the agent got checked the last time we can find the beginning of a path for every agent.

If an agent arrives at a location where not yet an agent proxy exists, one is created (Figure 10a). As soon as the agent migrates to another location, the destination (being part of the agent trail) is stored in the proxy together with the *time to live* (Figure 10b).

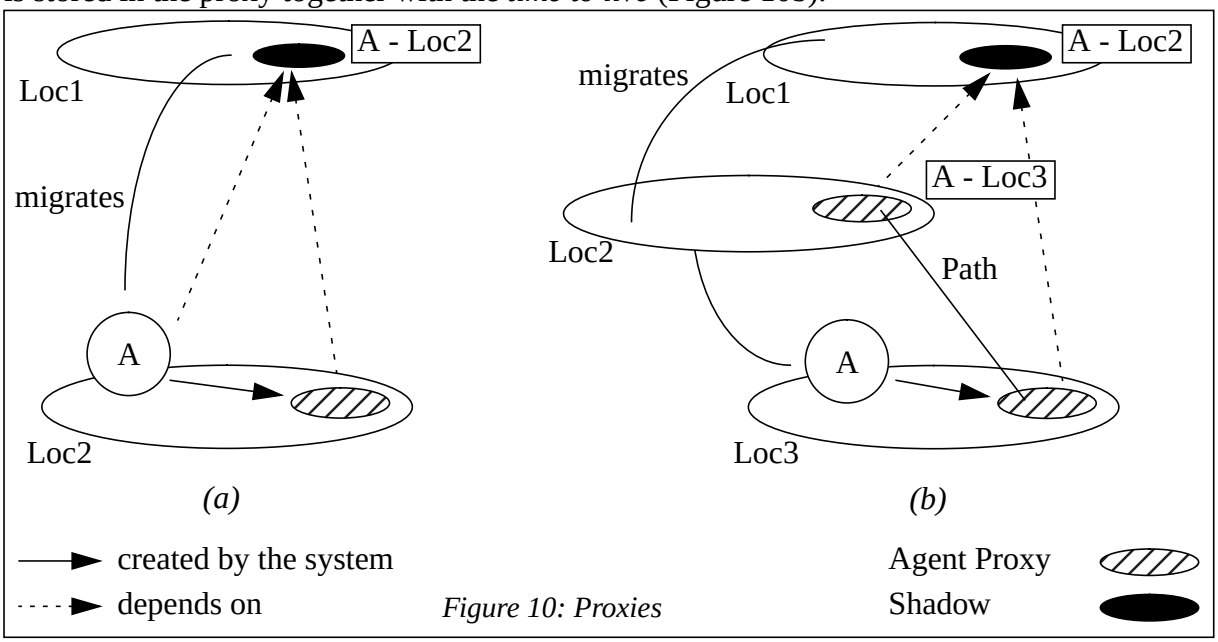

Figure 10: Proxies

When the end of the *time to live* is reached, the agent's shadow gets a request for extending the agent's life, and thus the new location of the agent is made known to the shadow (Figure 11a). The path stored in the different agent proxies along the agent's way is now superfluous and can

be removed (Figure 11b) using the knowledge about the *time to live*. An entry can also be removed if the agent migrates back to this location (this simply cuts the now circular path short).

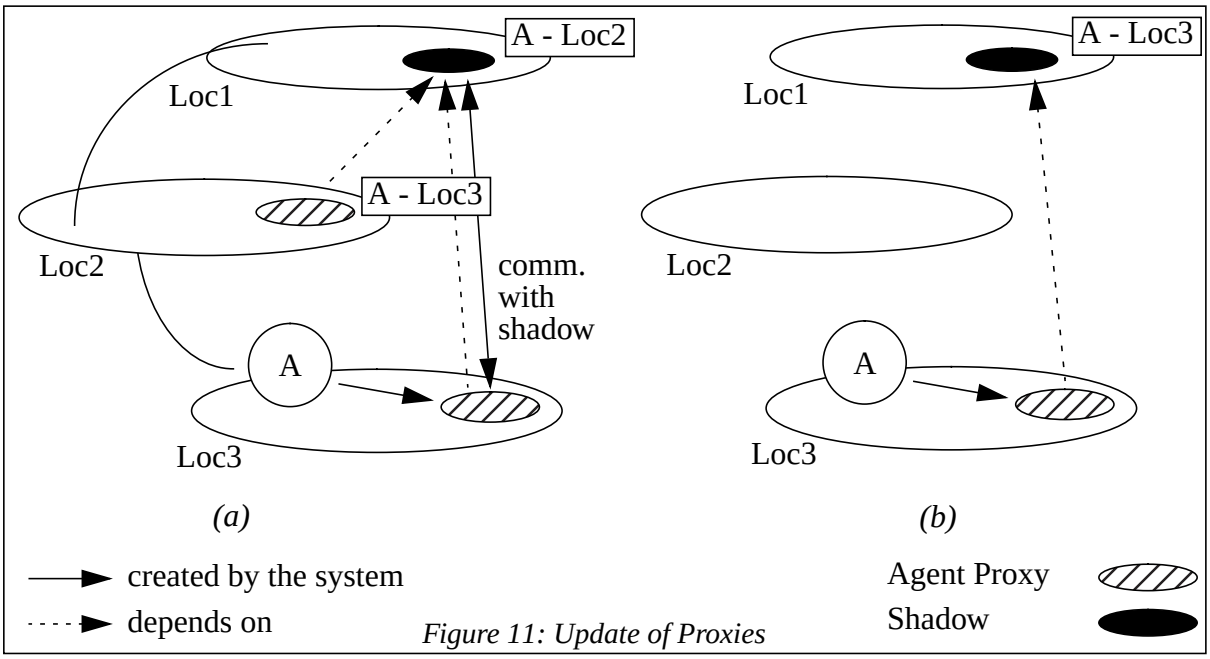

Figure 11: Update of Proxies

A proxy exists exactly as long as there is an entry in it. As soon as the agent proxy contains no entries, it can be removed as well. This is especially helpful, if the agents are actively terminated. In that case, all entries are removed from the proxy, allowing the system to delete the proxy also.

7.2 The Protocol

As before, the protocol as a whole can be found in the appendix (appendix A.3.3).

The location on which the agent resides, decrements in regular intervals the *time to live* of the agent. If the *time to live* of the agent is 0, a message is sent back to the home location of the shadow, containing the id of agent and shadow. At the same time a timer is started (the reaction to that is discussed below) and the agent enters the check phase. To allow greater flexibility each shadow (and thus each associated agent) can have a timeout of its own. But this allows for another loophole by setting a very long timeout. But it can be fixed by introducing a second timeout for each location. The timeout chosen is the minimum of agent timeout and location timeout.

If an agent arrives at a location, the list of proxies is examined if a proxy already exists. If not, a new one is created. The agent is add-

```
Regular Intervals:
    for each agent
        agent.timeToLive - -;
        if (agent.timeToLive == 0)
            sendCheck(agent.shadowHome, this,
                      agent.shadowId, agent.id);
            startTimer(min(location.TimeOut,agent.timeOut),
                       agent.proxy, agent);

onArrival(agent)
    proxy = proxyList.find(agent.shadowId);
    if(proxy == null)
        proxy = new Proxy(agent.id, agent.timeToLive,
                          agent.shadowHome, this);
        proxyList.add(proxy);
    else
        proxy.add(agent.agentId, agent.timeToLive);
    agent.proxy = proxy;
    agentList.add(agent);
    agent.start();

onLeaving(agent, target)
    if (agent.timeToLive > 0)
        agentList.remove(agent);
        agent.proxy.setTarget(agent.id, target));
        startTimer( agent.timeToLive + agent.timeOut,
                    agent.proxy, agent.id);
        SendAgent(target, agent);
```

Figure 12: System Methods

ed to the proxy, and the agent gets a reference on it. As soon as an agent wants to leave, its *time to live* is checked. This is done to prevent an agent who is in the check phase to migrate. If it is not in the check phase, the information in the proxy is updated to point to the target location. At the same time a timer is started that removes the path after the sum of remaining *time to live* and timeout (see Figure 12).

The check message sent is received by the home location of the shadow. First a timer is stopped that has been started the last time the *time to live* has been sent back to the agent. This allows to detect agents that have been terminated (see below). The *time to live* is requested from the shadow responsible, and if greater 0 is sent back to the requesting agent.

```
receiveCheck(from, shadowId, agentId)
    stopTimer(agentId);
    shadow = shadowList.find(shadowId);
    timeToLive = shadow.timeToLive(from, agentId);
    if (timeToLive > 0)
        startTimer( timeToLive
                    + shadow.getTimeOut(agentId),
                    shadow, agentId);
        sendAllowance(from, agentId, timeToLive);
receiveAllowance(agentId, timeToLive)
    stopTimer(agentId);
    agent = agentList.findAgent(agentId);
    agent.timeToLive = timeToLive;
    proxyList.setTime(agentId, timeToLive);
```

Figure 13: The Check Phase

As soon as the message is received, the timer for the timeout is stopped, and the agent's *time to live* is set (see Figure 13). This ends the agent's check phase and allows it to migrate again.

The shadow can decide on a case-by-case basis, if the agent's life time is extended and by what interval. In Figure 14 we present an example policy, that for all of the agents returns the same *time to live*. This method checks first if an agent proxy already exists for this agent (in case a newly created agent contacts the shadow), updates the information about the location of the agent, and returns the *time to live*. The shadow is also called if it has been detected (via the timeout), that an agent has been terminated. The simplest policy is to remove the related entry from the list.

```
timeToLive(from, agentId)
[implement policy]
    agentProxy = listOfProxies.find(agentId);
    if(agentProxy != null)
        agentProxy.target = from;
    else
        agentProxy = new AgentProxy(from, agentId,
                                    timeToLive);
    return agentProxy.timeToLive;
remove(agentId)
[implement policy]
    agentProxy = listOfProxies.find(agentId);
    listOfProxies.remove(agentProxy);
```

Figure 14: Methods in the Shadow Object

What remains to discuss is the reaction to the different timeouts (see Figure 15). One possible reaction to the timeout of the check message has been sketched out above. Here we present a simple alternative, the agent is removed at once.

```
onTimer(proxy, agent)                    // check timeout
[implement policy]
    agentList.remove(agent);
    proxy.remove(agentId);
    if(proxy.entries() == 0)
        proxyList.remove(proxy);
onTimer(proxy, agentId)                  // path is redundant
[implement policy]
    proxy.remove(agentId);
    if(proxy.entries() == 0)
        proxyList.remove(proxy);
onTimer(shadow, agentId)                 // agent terminated
    shadow.remove(agentId);
```

Figure 15: System: Reaction to Timeouts

The next timeout affects the paths. As soon as an agent migrates, the path segment pointing to its new location is created, and a timer started. As soon as this timer ends, we know that the path information in the shadow itself has been updated, and this part of the path can be removed. The last timed method is called if an agent has not tried to contact the shadow for the sum of *time to live* and timeout. It calls the shadow method (see Figure 14) to react to it.

7.2.1 Finding Agents

If we want to terminate an agent, we have to find it first. This can be done with the help of the information stored in the agent proxies. If the agent is in the local list of active agents, it is already found. If not, the related agent proxy is searched. If it is not found, an error is returned. If it is found, then a find request is sent to the target found in the proxy. At the target location the list of active agents is again examined. If the agent is found, a success message is sent back. If not, the agent proxy is searched. If no proxy exists, an error is sent back. Otherwise, the message is sent onward. This is repeated until the agent is found or the path ends (see Figure 16).

```
find(agentId)
    if (agentList.find(agentId) != null)
        return(this);
    agentProxy = shadowList.find(agentId);
    if(agentProxy != null)
        sendFind(agentProxy.target, this, agentId);
    else
        return(notFoundError);

receiveFind(searcher, agentId)
    if (agentList.find(agentId) != null)
        sendFound(searcher, this, agentId);
    if(proxyList.find(agentId) != null)
        sendFind(proxy.target, searcher, agentId);
    else
        sendError(searcher, notFoundError, agentId);

receiveFound(from, agentId)
    return(from);

receiveError(error, agentId)
    if (error == notFoundError)
        return(error);
```

Figure 16: Finding Agents

7.3 Mobile Shadows

The Idea

In cases where many of the agents depending on a shadow move somewhere far away (i.e. communication costs are high), every one of the agents has to contact the shadow independently with high cost. If the migration behaviour is known in advance, the shadow can be placed near these agents to reduce the communication cost. But in many cases the behaviour is not known in advance, or the group moves as a whole from area to area (e.g. from one organization to another). It would be much better if the shadow moved with the agents. If a shadow can be moved, possible policies where to place the shadow might be at a location where the communication cost to all dependent agents would be lowest, or to place the shadow where one crucial agent is situated. If then the location goes down (e.g. crashes), both shadow and agent would not be reachable, and the other dependent agents would be terminated. While in one case the shadow would have to be persistent, in the other case it would have to be transient to implement the policy.

To move a shadow two problems have to be dealt with. The first is that the agents depending on the shadow have somehow to be notified about the new location of the shadow. The second is that the application still has to be able to reach the shadow, e.g. in case it wants to terminate the agents. Both problems can be solved similar to the approach used with the agent proxies.
When a shadow moves, a copy of the shadow stays behind. Thus over time a shadow path is built. By contacting the copy at the home location in regular intervals this path can be cut short. As alternative to intervals at which to cut the path short, a maximum path length would be suitable. But using a maximum path length adds communication along the path, because after the maximum path length has been reached the shadow copies along the path have to be notified that they are no longer needed.
Now, when an agent requests a new time to live, it can happen that the shadow has moved somewhere else. In this case, the request is sent to the new location of the shadow. If the shadow already moved again, the request is forwarded along the path of shadow copies until the shadow

is reached. The shadow sends a new grant back to the agent together with its new location. The next time the agent sends its request directly to the new location.

The copies can be removed as soon as the path is no longer needed and no agent still has the reference to a copy of the shadow. Thus the maximum of agent and shadow time to live is the maximum time the copy has to be hold. One exception has to be made. The first copy, that stays at home, cannot be removed. Only if the shadow returns to the home location, this can be done.

The Protocol

The whole protocol can be found in appendix A.3.5. We first examine the shadow part of the protocol.

Moving the shadow to another location creates a path to the target, and starts a timer. After the timeout of this timer the path has to be deleted. We simplify by choosing the same *time to live* for shadow and agents. The path is created by leaving a shadow proxy behind. Removing the shadow is done by sending a message along the path (see Figure 17).

```
move(target)
    if (timeToLive != 0)
        sendShadow(target, this);
        if(currentLocation != null)    // part of the path
            pathTimeOut = timeToLive + timeOut;
            startTimer(pathTimeOut, shadow);
        currentLocation = target;

terminateShadow()
    if (currentLocation != null)// shadow moved out
        sendTerminate(currentLocation, id);
    delete(this);
```

Figure 17: Additional Shadow Methods

Each shadow gets a *time to live*, after which it must contact its home location. This time does not necessarily have to be the same as for the agents. In regular intervals this *time to live* is decremented. As soon as the shadow's time to live is 0, the shadow enters the check phase. A message containing the shadow id and its current location is sent to the home location and a timer is started (see Figure 18).

```
Regular Intervals:
[agent related part stays the same]
    for each shadow
        if (shadow.homeLocation != location.name())
            shadow.timeToLive--;
            if (shadow.timeToLive == 0)
                sendCheck(shadow.homeLocation,
                          shadow.id);
                startTimer(shadow.timeOut, shadow);
```

Figure 18: Extended System Methods: Regular Intervals

The check message contains the new location of the shadow. If the shadow copy at home still exists, it is updated and the *time to live* is sent back. If the answer is not received until the timeout, the shadow is removed. As soon as it is received, the timer is stopped and the *time to live* is set (see Figure 19). The shadow copies creating the path between home location and shadow get a similar timeout after the sum of *time to live* and communication timeout. At that point the path is redundant and can be removed (see below). This way the path created by the shadow is cut short in regular intervals.

```
onTimer(shadow)      // this path seg. is redundant
    shadowList.remove(shadow);

receiveAllowance(shadowId, timeToLive)
    shadow = shadowList.find(shadowId);
    stopTimer(shadow);
    shadow.timeToLive = timeToLive;

receiveCheck(from, shadowId)
    shadow = shadowList.find(shadowId);
    if(shadow != null)
        shadow.currentLocation = location;
        sendAllowance(from, shadowId,
                      shadow.timeToLive);
```

Figure 19: Additional System Methods: Checking the shadow

If the shadow comes back to its home location, the copy of the shadow is replaced by the original (see appendix A.3.5 for details).

In the basic protocol the check message is sent to the shadow's home location. Now it is sent to the location from which the last *time to live* message has been received. This is done by storing it in an additional attribute containing the agent shadow's location. If the shadow moves between two such messages, the check message is sent to a shadow proxy (somewhere on the path) instead of the original. The shadow proxy now forwards this check message along the path. The original, upon receiving the message, sends back the *time to live* and its own location. The path is superfluous as soon as the shadow's location is known at the home location **and** no agent still references a part of it. Since we chose the shadow and agent *time to live* to be the same, the path timeout is the sum of *time to live* and timeout (see Figure 20).

```
receiveCheck(from, shadowId, agentId)
    stopTimer(agentId);
    if(currentLocation != location.name())
        sendCheck(currentLocation, from,
                shadowId, agentId);
    else
        shadow = shadowList.find(shadowId);
        timeToLive =
                shadow.timeToLive(from, agentId);
        if (timeToLive > 0)
            startTimer(timeToLive
                    + shadow.getTimeOut(agentId),
                    shadow, agentId);
            sendAllowance(from, location.name(),
                    agentId, timeToLive);

receiveAllowance( shadowLocation, agentId,
                timeToLive)
    stopTimer(agentId);
    agent = agentList.findAgent(agentId);
    agent.timeToLive = timeToLive;
    agent.shadowHome = shadowLocation;
    proxyList.setTime(agentId, timeToLive);
```

Figure 20: Changed System Methods: Extending the agent's life

Together with sending back the *time to live* to the agent the shadow starts a timer. If after this timeout the agent did not send a check message, the shadow knows that the agent has terminated. But if the shadow has moved on in the meantime, this information only reaches a proxy and has to be sent after it. Thus a message is sent along the path containing the information that the agent has terminated. Every proxy sends the information onward until it reaches the shadow. Here the agent entry is removed (see Figure 21).

```
onTimer(shadow, agentId)// agent is terminated
    shadow.remove(agentId);
    if (shadow.currentLocation != location.name() )
        sendRemoved( currentLocation, shadowId,
                agentId);

receiveRemoved(shadowId, agentId)
    shadow = shadowList.find(shadowId);
    if(shadow != null)
        if(shadow.currentLocation != location.name())
            sendRemoved( currentLocation, shadowId,
                    agentId);
        else
            shadow = shadowList.find(shadowId);
            shadow.remove(agentId);
```

Figure 21: Changed System Methods: Extending the agent's life

7.4 Optimizing the Communication

As soon as more than one agent belongs to a shadow, optimizations of the communication are possible. Three optimizations exist:

- If two agents belonging to the same shadow come to the same location, the one with the lower time to live gets the remaining time of the other one. This works with an arbitrarily large number of agents on a location and happens expediently at the arrival of a new agent. All others will at that time have the same remaining interval already, so it is sufficient to check one of the residing agents against the new one.
- If an agent's shadow has been checked, then this information also gets transferred to all other agents on the same location.
- The combination of shadow, proxies and trail creates a spanning tree that follows the agents movements with the shadow as the root. This tree can be optimized by simply using common trails for the parts of the trails that are the same for different agents. This effectively reduces the number of messages that flow without changing the functionality. Furthermore,

the agents on nodes along the tree can be updated. The mechanism is the same as the one used in multicast protocols.

7.4.1 Terminating agents actively while using the optimizations

The proxies allow to find an agent, e.g. to terminate it actively. But with all of the mentioned optimizations the trail can be lost. This can happen if an agent gets additional *time to live* from another agent, and the path assuming the original *time to live* is removed. The optimizations make it impossible to terminate a specific agent.

The interesting point though is that it doesn't matter for the termination of the whole group of agents. If the termination message is sent to all known proxies, then these proxies forward the termination message along all of the paths they are part of. Ultimately this termination message reaches all of the agents, even those no longer directly known to the shadow. The path segment for an agent exists exactly for the then current *time to live* of the agent. So if it got additional time, than at that location the agent proxy holds the path from that location for that remaining time. Every time an agent gets additional time from another agent, there exists a valid path for that other agent. So, by first following the path to the other agent, and then the still valid path to our agent, every agent gets the termination message.

This way, all of the mentioned optimizations can be used without compromising functionality.

8 Comparison of the Protocols

In this section we compare the energy concept, the path concept, and the shadow protocol in the light of the requirements defined in section 4.

In table 1 the capacity of the different concepts to conform to the requirements is presented. It can easily be seen that all of the protocols conform to the basic requirements of distributed systems. The shadow concept trades communication cost for flexibility and functionality compared to both energy and path concept.

Table 1: The different Concepts in the light of the Requirements

Requirement	Energy Concept	Path Concept	Shadow Concept
Restrictions concerning the communication model	No	No	No
Restrictions concerning the order of messages	No	No	No
Acknowledgement of every single message	No	No	No
Restrictions concerning the topology of the agent system	No	No	No
Global time necessary	No	No	No
Freezing necessary to obtain necessary knowledge	No	No	No
Well-defined fault semantics	Yes	No	Yes
Communication cost when terminating	low	high	middle
Communication cost otherwise	middle	low	middle
The algorithm has loopholes	No	No	No

Table 1: The different Concepts in the light of the Requirements

Requirement	Energy Concept	Path Concept	Shadow Concept
An agent can create other agents	Yes	Yes	Yes
The system checks if an agent is an orphan	Yes	No	Yes
An agent has a limited life span	Yes	No	Yes
It is possible to extend the life span of an agent	Yes	-	Yes
Application has to be reachable to extend life span	Yes	-	Yes
Type of Termination	Passive	Active	Passive/Active

The shadow concept is less fault sensitive than the path concept, because the agent proxy paths are pruned in regular intervals. This adds communication costs. Depending on the usage of agents this can even be substantial compared to the energy concept (e.g. an agent that only wakes every other week). But in normal applications (e.g. information retrieval or electronic commerce scenarios) the communication cost will be alike. When actively terminating, the communication cost with the shadow concept is lower than with the path concept. The energy concept cannot actively terminate. In the case of passive termination, for the shadow concept upper bound for the time until all agents are terminated, can be given. This is not possible with the energy concept. Furthermore the shadow concept is very flexible and can easily adopted to different policies. Short-lived agents can be created without additional communication (as long as the life of the agent is shorter than the *time to live* of the creating agent). The shadow concept provides orphan detection and termination as a system service that is transparent to the programmer. This greatly simplifies the implementation of agents using this concept.

9 Related Work

In the area of distributed systems the term termination has a slightly different meaning. But many mechanisms have been developed in that area to determine if distributed processes have terminated (see e.g. [Mattern89]).

The problem of orphan detection in agent systems has some similarities to the problem of distributed garbage collection (see e.g. the work on Stub Scion Pair Chains [Shapiro92]). While the distributed garbage collection can already rely on some kind of relationship, e.g. through pointers or references, and thus has not to artificially create a structure, there are some similar problems to that in the area of mobile agents (see e.g. [BagPiu96]).

In [BauRad97] the use of a group concept for, among other things, termination and orphan detection purposes has been discussed.

10 Conclusion and Future Work

In this paper we discussed three concepts, the energy concept, the path concept, and the shadow concept. While the energy concept is a concept for orphan detection and passive termination, the path concept is a concept for finding agents and for actively terminating them. By combining both concepts in the shadow concept, we eliminate the disadvantages, and at the same time maintain the advantages of both concepts. Nevertheless, the shadow concept has still some dis-

advantages: it introduces additional communication into the system and resources are bound (memory) to store the different path information. But the advantages outweigh the disadvantages by far: the mechanism is robust against malicious or faulty agents, the path information is updated without additional communication costs (no outdated path information exists), and the time until all agents are terminated in the worst case, can be determined exactly. All presented protocols have been implemented in our agent system Mole (for a description of Mole see [StBaHo96, BaumEA97]).

In the future we will try to add the energy concept to the shadows on still another level. Instead of terminating the agent after a certain delay, the agent will get an amount of energy to use until the shadow can be connected again. This would allow the agent to decide on its own if it goes on with its work, either hoping the shadow is reachable before its energy is used up, or to sleep and play it safe. By using the energy concept only when a failure occurs (network partitioning or system crash), the agent gets more autonomy without compromising the shadow concept.

Fault tolerance is another area that has to be investigated in detail. The presented mechanism allows for short time network partitioning and system faults, but does not cope well with long time faults. We will investigate in which way the shadow concept can be made fault resilient by replication of the control structures.

The shadow structure could additionally hold some rules that determine if the goal of the agents belonging to this shadow can still be accomplished, e.g. the so-called AND-groups and OR-groups. In an AND-group all agents have to succeed in their task for the whole group to be successful. In an OR-group only one of the agents has to succeed for the group to be successful. As soon as the success or failure of the group can be determined, the other participating agents can either be terminated actively by sending a termination message, or passively by simply removing the shadow. This would however suppose a means to communicate additional information to the shadow: the exit state of the participating agents (success or failure).

We will also try to model termination and orphan detection, of single agents and of agent groups, with distributed event services, as defined e.g. by OMG's event service specification (see [BaumEA97, BauRad97] for details).

In the area of distributed systems and in the area of distributed garbage collection many mechanisms have been developed. We will investigate further, if these mechanism might be put to use in the area of mobile agents.

Acknowledgements: The described protocols have been implemented mostly by Eric Jochum [Jochum97] and Matthias Zepf [Zepf96]. The comments of Fritz Hohl and Markus Straßer greatly improved the quality of this paper.

A The Protocols

In this appendix the protocols are listed as a whole. Each of them is presented as methods in a pseudo object-oriented fashion. Some basic object types, e.g. lists are assumed existing. Methods can be called asynchronously, e.g.

- **startTimer**(*time*, *agentId*) will call a method **onTimer**(*agentId*) after *time*.
- **sendMessage**(*location*, „*Hello*“) is sent to **location** and calls **receiveMessage**(„*Hello*“).

Methods bodies containing [implement policy] can be used to implement a specific policy, e.g. for reacting to a message asking for more energy (in the energy concept). The presented implementation is one of the possible policies.

A.1 Energy Concept: The Protocol

Location: Methods

Regular Intervals:
```
for each agent
    agent.energy = agent.energty +agent.livingCost;
    if (livingAgents.find(agent) != null)
        checkLowEnergy(agent);
    if(agent.energy == 0) remove agent;
```

onArrival(agent)
```
receiveAgent(agent);
wake(agent);
agent.start();
```

onLeaving(agent, target)
```
livingAgents.remove(agent);
SendAgent(target, agent);
```

SystemMethod(method, agent)
```
if agent.energy < method.cost
    throw NotEnoughEnergy;
agent.energy = agent.energy - method.cost;
method.callMethod(agent);
checkLowEnergy(agent);
```

receiveEnergy(agentId, amount)
```
agent = getAgent(agentId);
agent.energy = agent.energy + amount;
agent.receiveEnergy();
if (agent.energy > agent.lowEnergy)
    agent.refilling = false;
    wake(agent);
```

receiveMessage(„energyLow“, agentId)
```
[implement policy];
```

checkLowEnergy(agent)
```
if (agent.energy < agent.lowEnergy
   AND agent.refilling == false)
    agent.energyLow();
    agent.refilling = true;
```

sleep(agent)
```
livingAgents.remove(agent);
agent.livingCost = location.sleepCost();
```

wake(agent)
```
livingAgents.add(agent);
agent.livingCost = location.livingCost();
```

Needed Objects

Object Method
```
Method callMethod(Agent);
Attribute cost:Integert;
...
```

Object Agent
```
Method energyLow();
Method receiveEnergy(Amount);
Attribute id:AgentId;
Attribute maxEnergy:Energy;
Attribute energy:Energy;
Attribute lowEnergy:Energy;
Attribute refilling:boolean;
Attribute livingCost:Energy;
Attribute home:LocationName;
...
```

Agent: Methods

EnergyLow()
```
[implement policy]
    sendMessage(home, “energyLow”, id);
    sleep();
```

ReceiveEnergy(amount)
```
[implement policy]
```

A.2 Paths: The Protocol

A.2.1 Basic Protocol

Needed Objects

```
Object PathProxy
    Attribute id:AgentId;
    Attribute target:LocationName;
Object Agent
    Attribute id:AgentId;
    Attribute home:LocationName;
    ...
```

Location: Methods

```
onArrival(agent)
    agentList.add(agent);
onLeaving(agent, target)
    agentList.remove(agent);
    pathProxyList.add(new PathProxy(agent.id, target));
    SendAgent(target, agent);
find(agentId)
    if (agentList.find(agentId) != null)
        return(this)
    pathProxy = pathProxyList.find(agentId);
    if(pathProxy != null)
        sendFind(pathProxy.target, this, agentId);
    else
        return(notFoundError);
receiveFind(searcher, agentId)
    if (agentList.find(agentId) != null)
        sendFound(searcher, this, agentId);
    if(pathProxyList.find(agentId) != null)
        sendFind(pathProxy.target, searcher, agentId);
    else
        sendError(searcher, notFoundError, agentId);
receiveFound(from, agentId)
    return(from);
receiveError(notFoundError, agentId)
    return(error);
```

A.2.2 Shortening Paths

Location: Additional and extended Methods

```
receiveFound(from, agentId)
    pathProxy = pathProxyList.find(agentId);
    sendMessage(pathProxy.target, shortenPath,
                    agentId);
    pathProx.target = from;
    return(from);
receiveError(from, error, agentId)
    if (error == notFoundError)
        shortenPath(from, agentId);
        return(error);
shortenPath(target, agentId)
    pathProxy = pathProxyList.find(agentId);
    if(target != this AND pathProxy != null)
            sendMessage(pathProxy.target, shortenPath, agentId);
            pathProxyList.remove(pathProxy);
```

A.3 Shadows: The Protocol

A.3.1 Objects needed

Needed Objects

Object Shadow
Method timeToLive(AgentId);
Method remove(AgentId);
Attribute listOfProxies:List of AgentProxy;
Attribute timeToLive:Integer;
Attribute timeOut:Integer;
...

Object AgentProxy
Attribute id:AgentId;
Attribute timeToLive:Integer;
Attribute target:LocationName;

Object Proxy
Attribute agents:List of AgentProxy;

Object Agent
Attribute id:AgentId;
Attribute timeToLive:Integer;
Attribute timeOut:Integer;
Attribute proxy:Proxy;
Attribute shadowId:ShadowId;
Attribute shadowHome:LocationName;
Attribute home:LocationName;
...

A.3.2 Methods in the Object Shadow

Shadow

timeToLive(from, agentId)
[implement policy]
```
agentProxy = listOfProxies.find(agentId);
if(agentProxy != null)
    agentProxy.target = from;
else
    agentProxy = new AgentProxy(from, agentId, timeToLive);
return agentProxy.timeToLive;
```

remove(agentId)
```
agentProxy = listOfProxies.find(agentId);
listOfProxies.remove(agentProxy);
```

A.3.3 Basic Protocol with Proxies

Location: Methods

Regular Intervals:

```
for each agent
    agent.timeToLive - -;
    if (agent.timeToLive == 0)
        sendCheck(agent.shadowHome, this, agent.shadowId, agent.id);
        startTimer(min(localTimeOut, agent.timeOut), agent.proxy, agent);
```

onArrival(agent)

```
proxy = proxyList.find(agent.shadowId);
if(proxy == null)
    proxy = new Proxy(agent.id, agent.timeToLive, agent.shadowHome, this);
    proxyList.add(proxy);
else
    proxy.add(agent.agentId, agent.timeToLive);
agent.proxy = proxy;
agentList.add(agent);
agent.start();
```

onLeaving(agent, target)

```
if (agent.timeToLive > 0)
    agentList.remove(agent);
    agent.proxy.setTarget(agent.id, target));
    startTimer(agent.timeToLive + agent.timeOut, agent.proxy, agent.id);
    SendAgent(target, agent);
```

onTimer(proxy, agent) **// called if agent has not been sent allowance in time**

[implement policy]

```
agentList.remove(agent);
proxy.remove(agentId);
if(proxy.entries() == 0)
    proxyList.remove(proxy);
```

onTimer(proxy, agentId) **// called if the time for the proxy path has ended**

```
proxy.remove(agentId);
if(proxy.entries() == 0)
    proxyList.remove(proxy);
```

onTimer(shadow, agentId) **// called if the agent has been killed due to timeout**

```
shadow.remove(agentId);
```

receiveAllowance(agentId, timeToLive)

```
stopTimer(agentId);
agent = agentList.findAgent(agentId);
agent.timeToLive = timeToLive;
proxyList.setTime(agentId, timeToLive);
```

receiveCheck(from, shadowId, agentId)

```
stopTimer(agentId);
shadow = shadowList.find(shadowId);
timeToLive = shadow.timeToLive(from, agentId);
if (timeToLive > 0)
    startTimer(timeToLive + shadow.getTimeOut(agentId), shadow, agentId);
    sendAllowance(from, agentId, timeToLive);
```

createAgent(creatingAgent, AgentClass, parameterList)

```
newAgent       = new AgentClass(parameterList);
newAgent.id    = createId();
newAgent.timeToLive= creatingAgent.timeToLive;
newAgent.timeOut= creatingAgent.timeOut;
newAgent.shadowId= creatingAgent.shadowId;
newAgent.shadowHome= creatingAgent.shadowHome;
newAgent.home= this.name;
onArrival(newAgent);
```

A.3.4 Finding Agents

Location: Methods

find(agentId)
```
if (agentList.find(agentId) != null)
    return(this);
if(shadowList.find(agentId) != null)
    sendFind(proxy.target, this, agentId);
else
    return(notFoundError);
```

receiveFind(searcher, agentId)
```
if (agentList.find(agentId) != null)
    sendFound(searcher, this, agentId);
if(proxyList.find(agentId) != null)
    sendFind(proxy.target, searcher, agentId);
else
    sendError(searcher, notFoundError, agentId);
```

receiveFound(from, agentId)
```
return(from);
```

receiveError(error, agentId)
```
if (error == notFoundError)
    return(error);
```

A.3.5 Mobile Shadows

Shadow: Additional Attributes

Object Shadow
```
Method move(LocationName);
Attribute currentLocation:LocationName;
Attribute homeLocation:LocationName;
Attribute timeToLive:Integer;
```

Shadow: Additional Methods

move(target)
```
if(timeToLive != 0)
    sendShadow(target, this);
    if(currentLocation != null)// this is not the first time, we are part of the path
        startTimer(timeToLive + timeOut, shadow);
    currentLocation = target;
```

terminateShadow()
```
if (currentLocation != null)// the shadow has moved out
    sendTerminate(currentLocation, id);
delete(this);
```

Location: Additional and Extended Methods

Regular Intervals:
```
for each agent
    agent.timeToLive - -;
    if (agent.timeToLive == 0)
        sendCheck(agent.shadowHome, this, agent.shadowId, agent.id);
        startTimer(min(localTimeOut, agent.timeOut), agent.proxy, agent);
for each shadow
    if (shadow.homeLocation != location.name())// only if not at home location
        shadow.timeToLive--;
        if (shadow.timeToLive == 0)
            sendCheck(shadow.homeLocation, shadow.id);
            startTimer(shadow.timeOut, shadow);
```

```
onTimer(shadow, agentId)                          // called if the agent has been killed due to timeout
    shadow.remove(agentId);
    if (shadow.currentLocation != location.name() )
        sendRemoved(currentLocation, shadowId, agentId);

onTimer(shadow)
    shadowList.remove(shadow);                    // called when the shadow path can be removed

receiveAllowance(shadowLocation, agentId, timeToLive)
    stopTimer(agentId);
    agent = agentList.findAgent(agentId);
    agent.timeToLive = timeToLive;
    agent.shadowHome = shadowLocation;
    proxyList.setTime(agentId, timeToLive);

receiveAllowance(shadowId, timeToLive)
    shadow = shadowList.find(shadowId);
    stopTimer(shadow);
    shadow.timeToLive = timeToLive;

receiveCheck(from, shadowId, agentId)
    stopTimer(agentId);
    if(currentLocation != location.name())        // we are not the current shadow
        sendCheck(currentLocation, from, shadowId, agentId);
    else
        shadow = shadowList.find(shadowId);
        timeToLive = shadow.timeToLive(from, agentId);
        if (timeToLive > 0)
            startTimer(timeToLive + shadow.getTimeOut(agentId), shadow, agentId);
            sendAllowance(from, location.name(), agentId, timeToLive);

receiveCheck(from, shadowId)
    shadow = shadowList.find(shadowId);
    if(shadow != null)
        shadow.currentLocation = location;
        sendAllowance(from, shadowId, shadow.timeToLive);

receiveShadow(shadow)
    if(shadow.timeToLive != 0)
        if(shadow.homeLocation != location.name())
            shadow.currentLocation = location.name();
            shadowList.add(shadow);
        else                        // shadow comes back home
            shadowList.find(shadow.shadowId);
            shadowList.remove(orig_shadow);
            shadowList.add(shadow);
            shadow.currentLocation = null;

receiveRemoved(shadowId, agentId)
    shadow = shadowList.find(shadowId);
    if(shadow != null)
        if(shadow.currentLocation != location.name())
            sendRemoved(currentLocation, shadowId, agentId);
        else
            shadow = shadowList.find(shadowId);
            shadow.remove(agentId);
```

B References

[BaumEA97] J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, M. Straßer. “Communication Concepts for Mobile Agent Systems”, in Proc. Mobile Agents ‘97, Springer Verlag, 1997.

[BagPiu96]] A. Baggio, I. Piumarta. “Mobile host tracking and resource discovery”, in Proc. of the 7th ACM SIGOPS European Workshop”, 1996.

[BauRad97] J. Baumann, N. Radouniklis. „Agent Groups for Mobile Agent Systems“, in Proc. DAIS ’97, to appear.

[BaTsVi96] J. Baumann, C. Tschudin, J. Vitek. “Mobile Object Systems: Workshop Summary”, Workshop Proceedings for the 2nd Workshop on Mobile Object Systems, in Workshop Reader ECOOP ’96, d-punkt.verlag

[ComSte93] D. E. Comer, D. L. Stevens. “Internetworking with TCP-IP: 3. Client-server programming and applications”, Prentice-Hall, 1993.

[GenMag97] General Magic, “Odyssey Web Site”. URL: http://www.genmagic.com/agents/

[Goscin91] A. Goscinski. “Distributed Operating Systems - The Logical Design”, Addison-Wesley, 1991.

[Aglets97] IBM, “The Aglets Workbench”. URL: http://www.trl.ibm.co.jp/aglets/

[JulEA88] E. Jul, H. Levy, N. Hutchinson, A. Black. “Fine-Grained Mobility in the Emerald System”, ACM Transactions on Computer Systems, Vol. 6(1), P. 109-133, 1988.

[Jochum97] E. Jochum. “Design and Implementation of the Energy Concept for Mobile Agent Systems”, Student Thesis (in german), University of Stuttgart, 1997.

[Matter89] F. Mattern. “Distributed Algorithms” (in german), Springer Verlag, 1989.

[Mole97] “Mole Project Pages”. University of Stuttgart, http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html

[ShDiPl92] M. Shapiro, P. Dickman, D. Plainfossé. “SSP Chains: Robust, Distributed References supporting acyclic Garbage Collection”, Technical Report No. 1799, INRIA, Rocquencourt, Frankreich, 1992.

[StBaHo96] M. Straßer, J. Baumann, F.Hohl. “Mole - A Java Based Mobile Agent System”, in Workshop Reader ECOOP ’96, d-punkt, 1996.

[Tanenb96] A. S. Tanenbaum. “Computer Networks”, 3rd Edition, Prentice-Hall, 1996.

[Tanenb95] A. S. Tanenbaum. “Distributed Operating Systems”, Prentice-Hall, 1995.

[White94a] J. E. White. “Telescript Technology: The Foundation of the Electronic Marketplace”, General Magic, 1994.

[White94a] J. E. White. “Telescript Technology: Scenes from the Electronic Marketplace”, General Magic, 1994.

[Zepf96] M. Zepf. “Design and Implementation of a simple Orphan Detection Mechanism for a Mobile Agent System”, Student Thesis (in german), University of Stuttgart, 1996.