Universität Stuttgart
Fakultät Informatik

The Shadow Approach: An Orphan Detection Protocol for Mobile Agents

Joachim Baumann, Kurt Rothermel

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

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

The Shadow Approach: An Orphan Detection Protocol for Mobile Agents

Joachim Baumann, Kurt Rothermel

Bericht 1998/08
July 1998

The Shadow Approach: An Orphan Detection Protocol for Mobile Agents

Joachim Baumann, Kurt Rothermel

IPVR (Institute for Parallel and Distributed High-Performance Systems)
Breitwiesenstraße 20-22
D-70565 Stuttgart
EMail:Joachim.Baumann@informatik.uni-stuttgart.de

Abstract. Orphan detection 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 controlling mobile mobile agents and for orphan detection is presented.
The 'shadow' approach presented in this paper uses the idea of a placeholder (shadow) which is assigned by the agent system to each new agent. This defines an artificial relationship between agents and shadow. The shadow records the location of all dependent agents. Removing the root shadow implies that all dependent agents are declared orphan and eventually be terminated. We introduce agent proxies that create a path from shadow to every agent. In an extension of the basic protocol we additionally allow the shadow to be mobile.
The shadow approach can be used for termination of groups of agents even if the exact location of each single agent is not known.

1 Introduction

A mobile agent is regarded 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. [12], [4], [8], [14], [7], [6]). But until now no protocols exist for orphan detection in mobile agent systems.

Orphan detection in an agent system is 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 (at least in principle), 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 anymore, he wants to be able to terminate the computation to minimize the resulting cost. With an orphan detection mechanism the user simply declares the agents to be terminated as orphans. Orphan detection guarantees that the now useless agents can be determined by the system and ended, thus freeing the resources they have bound. In this paper we will present a new protocol, the shadow protocol, that allows both control of mobile agents and orphan detection. The paper is organized as follows: Sect. 2 presents our agent model. In Sect. 3 the shadow protocol is presented with different extensions and optimizations. Sect. 4 presents related work, and in Sect. 5 the conclusion and outlook is given.

2 The Agent Model

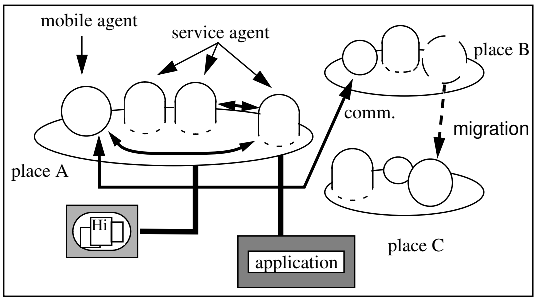

Fig. 1. The Agent Model

In this section we will give you a short overview of our agent model, that has been described in more detail in [12], [1] and [4]. Our model of an agent-based system - as many other models - is mainly based on the concepts of agents and places. Places provide the environment for safely executing local as well as visiting agents. An agent system consists of a number of (abstract) places, being the home of various services. Agents are active entities, which may move from place to place to meet other agents and access the places' services. 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 place can be derived from this name. It is independent of the agent's current place, i.e. it does not change when the agent moves to a new place. In other words, the applied identifier scheme provides location transparency. A place is entirely located on a single node of the underlying network, but multiple places may be situated on a given node. For example, a node may provide a number of places, each one assigned to a certain agent community, allowing access to a certain set of services or implementing a certain prizing policy. Places 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 place on this system is called *connected*. If a system is only part-time connected to the network, e.g. a user's PDA (Personal Digital Assistant), the place is called *associated*.

3 The Shadow Protocol

In this section we discuss the basic Shadow Protocol with its agent proxies, the extension that allows the shadows to be mobile, and discuss possible optimizations.

3.1 The Idea

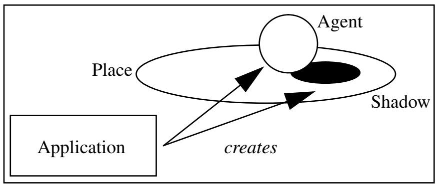

Fig. 2. The Creation of a Shadow

In the shadow concept each application creates one or more shadows, a data structure on a connected place. The place where the shadow is created does not necessarily have to run on the same host on which the creating application runs. Each agent created by the application depends on such a shadow (Fig. 2). As long as the shadow exists in the system, no contact of agents to the application itself or to the computer system on which the application runs is necessary. In regular intervals (called *time to live* or *ttl*) the system checks for each agent if the associated shadow still exists. If the shadow does no longer exist (because the application removed it), the agent is declared to be an orphan and is removed.

If an agent creates a new agent, the system assigns the to this new agent the shadow of the creating agent, and the same remaining *ttl* until the next check (Fig. 3). This assignment cannot be changed by the agents. Limiting the time span to the remaining *ttl* of the creating agent (and not to the original time interval) 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 *ttl* just before the life span of the old agent ends. If a place on which a shadow resides cannot be reached, the system tries to contact the place several times. If still the place cannot be reached, the shadow is presumed no longer existent and its associated agents are killed. The disadvantage of this approach is that regardless of what an agent does, it has to connect to its shadow's place in regular intervals. The advantage on the other hand is that we have a worst-case time bound for the termination of agents through removing the shadows. This upper bound is exactly the sum of *ttl* of the agents and the timeout for contacting.

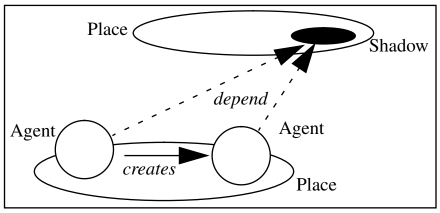

Fig. 3. Creating a New Agent

Until now the protocol only allows passive termination. By removing a shadow all dependent agents are declared orphans, and after the *ttl* it is guaranteed that all agents have been removed by the orphan detection. By adding the *path* concept to this protocol, we also allow active termination, i.e. termination of an agent while its *ttl* is greater 0. Agent proxies are structures at each place that keep track of the movement of all agents dependent of a specific shadow, thus creating a path leading to the agent. By storing the place at which the agent got checked the last time we can find the beginning of a path for every agent. Even if the path gets lost, the agent will contact the shadow after the *ttl*.

If an agent arrives at a place where not yet an agent proxy for this shadow exists, one is created (Fig. 4). As soon as the agent migrates to another place, the destination (being part of the path leading to the agent) is stored in the proxy together with the *ttl*.

When the end of the *ttl* is reached, the agent's shadow gets a request for extending the agent's life, and thus the new place of the agent is made known to the shadow (Fig. 5). The path entries stored in the different agent proxies along the agent's way is now superfluous and can be removed using the knowledge about the *ttl* stored in the proxy. An entry can also be removed if the agent migrates back to this place (this simply optimizes the now circular path by removing the loop).

An agent proxy contains, for a specific place, all path segments of agents belonging to the same shadow. It exists exactly as long as there is a path entry in it. As soon as the agent proxy contains no more entries, it can be removed as well. This is especially helpful if the agents are actively terminated, i.e. the system actively sends messages to terminate the agents as fast as possible. In that case, all entries are removed from the agent proxy, allowing the system to delete the proxy as well.

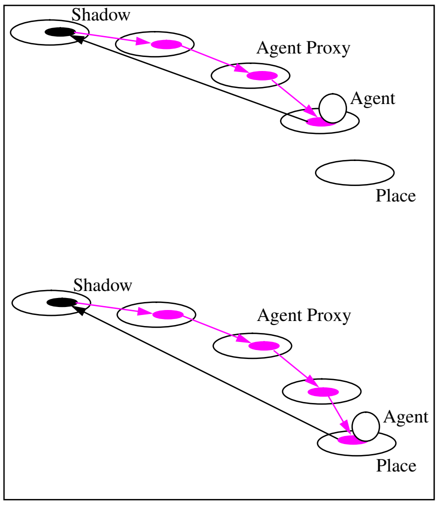

Fig. 4. Proxie Paths

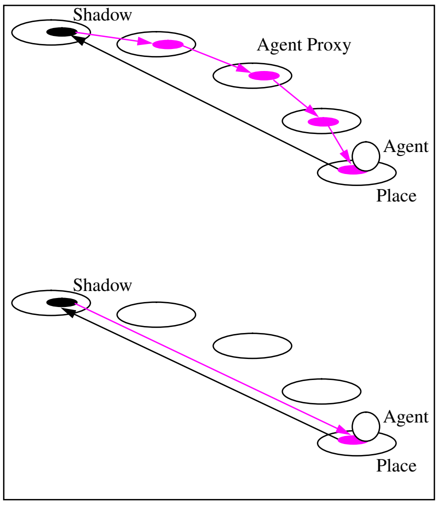

Fig. 5. Regular Update of Proxies

3.2 The Protocol

We will discuss the different parts of the protocol separately. The protocol is presented in an object-oriented pseudo-code notation.

The place on which the agent resides, decrements in regular intervals the *ttl* of the agent. As soon as the *ttl* of the agent is 0, a message is sent back to the home place of the shadow, containing the id of agent and shadow. At the same time a timer is started with a timeout, and the agent enters the *check phase* (Fig. 6). To allow greater flexibility each shadow (and thus the group of associated agents) can have a timeout of its own. This allows for a loophole by setting a very long timeout. But this can be corrected by introducing a per-place timeout. The timeout finally chosen is the minimum of agent timeout and place timeout.

```
Regular Intervals:
    for each agent
        agent.timeToLive - -;
        if (agent.timeToLive == 0)
            sendCheck(agent.shadowHome, current-
Place,
                    agent.shadowId, agent.id);
            startTimer(min(place.TimeOut,agent.time-
Out),
                        agent.proxy, agent);
onArrival(agent)
    agentproxy = proxyList.find(agent.shadowId);
    if(agentproxy == null)
        agentproxy = new Proxy(agent.id, agent.timeTo-
Live,
                        agent.shadowHome, current-
Place);
        proxyList.add(agentproxy);
    else
        agentproxy.add(agent.agentId, agent.timeTo-
Live);
    agent.proxy = agentproxy;
    agentList.add(agent);
    agent.start();
onLeaving(agent, target)
```

Fig. 6. System Methods

The check message is received by the home place of the shadow. First a timer is stopped that has been started the last time the *ttl* has been sent back to the agent. This allows to detect agents that have been terminated (see below). The *ttl* is requested from the responsible shadow, and if greater 0 is sent back by the system to the requesting agent. As soon as the message is received, the timer for the timeout is stopped, and the agent's *ttl* is set (see Fig. 7). This ends the agent's check phase and allows it to migrate again. When an agent arrives at a place, the list of agent proxies is searched for a proxy of that agent. If none exists, a new one is created, and the agent gets a reference on it. As soon as an agent wants to leave, its *ttl* 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 agent proxy is updated to point to the target place. At the same time a timer is started that removes the path after the sum of remaining *ttl* and timeout (see Fig. 6). The shadow can decide on a case-by-case basis if an agent's life time is to be extended, and by which interval.

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

Fig. 7. The Check Phase

```
timeToLive(from, agentId, shadowId)
[here an example policy is presented]
    shadowproxy = listOfProxies.find(shadowId
    agententry = shadowproxy.get(agentId)
    if( agententry != null)
        agententry.target = from;
    else
        agententry = new AgentEntry(from, agen
                            timeToLive);
        shadowproxy.add (agententry);
    return agententry.timeToLive;
remove(agentId)
[implement policy]
    agentproxy = listOfProxies.find(agentId);
```

Fig. 8. Methods in the Shadow Object

In Fig. 8 we present an example policy, that for all of the agents returns the same *ttl*. This method checks first if an agent entry 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 *ttl*. The shadow is also called if the system has detected (via the timeout), that an agent has been terminated. The simplest policy is to remove the related entry from the list. We now discuss the reaction to the different timeouts (see Fig. 9). One possible reaction to the timeout of the check message has been sketched out above. Here we present a

```
onTimer(proxy, agent)          // check timeout
[here an example policy is presented]
    agentList.remove(agent);
    agentproxy.remove(agentId);
    if(agentproxy.entries() == 0)
        proxyList.remove(agentproxy);
onTimer(agentproxy, agentId)// path redunda
[implement policy]
    agentproxy.remove(agentId);
    if(agentproxy.entries() == 0)
        proxyList.remove(agentproxy);
onTimer(shadow, agentId)    // ag. terminated
```

Fig. 9. System: Reaction to Timeouts

simple alternative; the agent is removed at once. 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 safely be removed. The last method is called if an agent has not tried to contact the shadow for the sum of *ttl* and timeout. In this case the agent has terminated. The shadow method (see Fig. 9) is called to react to it.

Finding Agents. If we want to actively terminate a specific 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 discovered, a *find request* is sent to the target found in the proxy. At the target place the list of active agents is again examined. If the agent is found, a success message is sent back. If not, the related agent proxy is searched again. If no proxy exists, an error is sent back. Otherwise, the message is sent on. This is repeated until the agent is found or the path ends (see Fig. 10).

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

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

receiveFound(from, agentId)
    return(from);

receiveError(error, agentId)
```

Fig. 10. Finding Agents

3.3 Mobile Shadows

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, resulting in unnecessarily high communication costs. If the migration behaviour is known in advance, the shadow can be placed in a way that reduces 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). In these cases it would be much better if the shadow moved with the agents. Possible policies where to place the shadow could be:

- at a place where the communication cost to all dependent agents would be lowest.
- where one agent important for the computation is situated. If the place becomes unavailable (e.g. crashes), both shadow and agent would not be reachable, and the other dependent agents would be terminated.

While in the first case the shadow would have to be persistent, in the second 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 shadow proxy stays behind. Thus over time a shadow path is built. By contacting the copy at the home place 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 as soon as the maximum path length has been reached the shadow proxies along the path have to be notified that they are no longer needed. A combination of these policies seems the most flexible.

Now, when an agent requests a new *ttl*, the shadow might already have moved somewhere else. In this case, the request is sent to the new place of the shadow. If the shadow already has moved again, the request is forwarded along the path of shadow proxies until the shadow itself is reached. The shadow sends a new grant back to the agent together with its new place. The next time the agent sends its request directly to the new place.

The shadow proxies can be removed as soon as the path is no longer needed and no agent still has the reference to a shadow proxy. Thus the maximum of agent and shadow *ttl* is the maximum time the proxy has to be hold. One exception has to be made though. The first proxy, that stays at home, cannot be removed as long as the shadow is elsewhere.

The Protocol. We first examine the shadow part of the protocol. Moving the shadow to another place creates a path to the target and starts a timer. After the timeout of this timer the path has to be deleted. The path is created by leaving a shadow proxy behind. Removing the shadow is done by sending a message along the path (see Fig. 11). Each shadow gets a *ttl*, after which it must contact its home place. This time is not necessarily the same as for the agents.

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

terminateShadow()
    if (currentPlace != null)    // shadow move
        sendTerminate(currentPlace, id);
```

Fig. 11. Additional Shadow Methods

In regular intervals this *ttl* is decremented. As soon as the shadow's *ttl* is 0, the shadow enters the check phase. A message containing the shadow id and its current place is sent to the home place and a timer is started (see Fig. 12). The check message for the shadow contains the new place of the shadow. If the shadow proxy at home still exists, it is updated and the *ttl* is sent back. If the answer

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

Fig. 12. Extended System Methods: Regular Intervals

is not received until the timeout, the shadow is removed (more complex reactions with retries can be chosen instead).

As soon as it is received, the timer is stopped and the *ttl* is set (see Fig. 13). The shadow proxies creating the path between home place and shadow get a similar timeout after the sum of *ttl* of the shadow, of the agent (see below) and the 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. If the shadow comes back to its home place, the shadow proxy is replaced by the original.

```
onTimer(shadow)      // this path seg. is redundan
    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.currentPlace = place;
        sendAllowance(from, shadowId,
```

Fig. 13. Additional System Methods: Checking the Shadow

In the basic protocol the agent check message is sent to the shadow's home place. Now it is sent to the place from which the last *ttl* message has been received. This is done by storing it in an additional attribute. 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 agent check message along the path. The original, upon receiving the message, sends back the *ttl* and its own place. The path is superfluous as soon as the shadow's place is known at the home place **and** no agent still references a part of it (see Fig. 14).

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

receiveAllowance(shadowPlace, agentId, timeToL
    stopTimer(agentId);
    agent = agentList.findAgent(agentId);
    agent.timeToLive = timeToLive;
```

Fig. 14. Changed System Methods: Extending the Agent's Life

Together with sending back the *ttl* 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 since the timeout is detected at a place and not inside the shadow, the information might only reach a proxy and not the shadow itself. In this case the shadow has to be informed. 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. Now the agent entry is removed (see Fig. 15).

```
onTimer(shadow, agentId)     // agent terminate
    shadow.remove(agentId);
    if (shadow.currentPlace != place.name() )
        sendRemoved( currentPlace, shadowId,
                     agentId);

receiveRemoved(shadowId, agentId)
    shadow = shadowList.find(shadowId);
    if(shadow != null)
        if(shadow.currentPlace != place.name())
            sendRemoved( currentPlace, shadowId
                         agentId);
        else
            shadow = shadowList.find(shadowId);
```

Fig. 15. Changed System Methods: Detecting Terminated Agents

3.4 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 place, the *ttl* of the one with the lower remaining time interval is set to the *ttl* of the other one. This works with an arbitrarily large number of agents on a place and happens conveniently at the arrival of a new agent.
- If an agent's shadow has been checked, then this information also gets transferred to all other agents belonging to the same shadow on the same place as the agent.
- The combination of shadow and agent proxies creates a spanning tree that follows the agents' movements with the shadow as the root. The tree can be optimized by simply using common paths for the parts of the paths 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 simulataneously.

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

The interesting point though is that this 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 current *ttl* of the agent. So if it got additional time, then at that place the agent proxy holds the path from that place for that remaining time. Every time an agent gets additional time from another agent, there exists a valid path to 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 for the group as a whole.

3.5 Fault Tolerance

Our fault model contains two types of failures, node failures (fail-stop) and network partitions. It is important to note that from the viewpoint of a node these failures are not distinguishable. By introducing a path of proxies the fault sensitivity of the protocol is increased. If only one of the nodes containing a proxy is not reachable, either through node failure or network partitioning, the path is broken. Different mechanisms have to be used for the two different kinds of paths. While in the case of a broken agent proxy path only one agent is no longer reachable until its *ttl* is 0, in the case of a broken shadow proxy path the agents trying to extend their life are threatened. The mechanism employed for the agent proxy paths has already been presented in Sect. 3, and is only discussed briefly. The mechanism used for shadow proxy paths has not yet been discussed in the protocol section and is examined in the following in detail.

Agent Proxy Path. By introducing the *ttl*, after which the agent has to contact the shadow's place, it is guaranteed that even if the path is broken, the new location of the agent can be identified after the *ttl* (as a worst-case bound), as long as either the network partition is short-term, or agent place and shadow place are in the same partition. If after the *ttl* (plus the timeout) the agent has not contacted the shadow, the shadow knows that the agent does not exist any longer (either because it has terminated or has been declared orphan and removed by the system).

Shadow Proxy Path. Two strategies are possible for dealing with a broken shadow proxy path. The first strategy does not change the characteristics of the protocol, but manages only short-term failures. It lets the last shadow proxy of the still-existing path try to contact the next shadow proxy again. The problem though is that the new *ttl* has to be sent to the agent before the system decides to terminate it.

The second strategy allows for longer failures but changes the worst-case bound for passive termination of the agents (the worst-case bound is $2ttl$ in this variant). If the last shadow proxy detects the break, it sends a new *ttl* back to the agent, but with the *home* place of the shadow as the new location. The new *ttl* is the minimum of the remaining shadow *ttl* and the agent *ttl*. If the shadow would have been removed, then the shadow proxy would know about it (and would have been removed as well). Thus the shadow still exists and it is correct to send the allowance. The home place of the shadow is sent instead of the location of the next shadow proxy in the path, to guarantee that the agent has a valid place to send the request for the next *ttl*. If the *ttl* of the agent is shorter than the remaining time of the shadow proxy path, then the next request will be sent along the same path (that hopefully is connected again). If the *ttl* of the path is shorter, then the agent will contact the home place of the shadow when the shadow itself has requested a new *ttl*. This means that the home place holds the new location of the shadow and forwards the request correctly.

4 Related Work

In the area of mobile agent systems the current research concentrates on the basic system support. But now that many different agent systems existing support the functionality needed to realize applications, mechanisms providing the functionality presented in this paper are essential. Thus the problem areas of orphan detection and termination of agents are beginning to evoke the interest of the research community. But apart from the mechanisms developed at the University of Stuttgart (see [5] describing a group concept or [2] discussing an energy concept and a path concept) no publications present similar functionality for mobile agent systems. However, in the area of distributed systems many algorithms exist that solve similar problems. The area of distributed algorithms, and especially distributed termination detection (in [9] and in [13] a discussion of many algorithms can be found) and distributed garbage collection (one example is the work on Stub Scion Pair Chains [11]), has to be seen as related work.

But two differences prevent the use of these algorithms for mobile agent systems. First of all, the fault model is different. The possibility of network partitions or node crashes does not exist in the fault model used for most distributed algorithms. Mobile agent systems explicitly include these faults in their fault model. Furthermore, the fault model supports the asynchrony of agents. The second difference is the autonomy of the "objects" in question that very much influences the processing model. A process (or object) in the distributed system area is not normally seen as autonomous. Here a process is seen as a cooperating part of a larger application. For a mobile agent the autonomy is one of the important prerequisites. This autonomy leads to the problem that a malicious agent might try to remove itself from the control by the system. These differences make it impossible to use the existing distributed algorithms in the area of mobile agent systems. It might be possible to use one such algorithm as the basis for a new design tailored to the needs of mobile agent systems. But the changes in the fault model and in the processing model effect so many changes in the algorithm itself that a *correct* transformation would be problematic at best. Nevertheless we believe that in principle it is possible to transform these algorithms correctly into algorithms that take the peculiarities of mobile agent systems into account. The key to this is an automatic transformation that, used on e.g. an algorithm for distributed garbage collection, turns it into a orphan detection and / or termination algorithm for mobile agent systems. An analogon to such an algorithm exists for the automatic transformation of termination detection algorithms into distributed garbage collection algorithms [10].

5 Conclusion and Future Work

In this paper we presented the shadow protocol. The shadow protocol has still some disadvantages: it introduces additional communication into the system and resources (memory) are bound 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. The presented protocol has been implemented in our agent system Mole (for a description of Mole see [12], [1], and [4]).

We will examine the area of fault tolerance in detail. The presented mechanism is robust against short time network partitioning and system faults, but does not cope well with lasting faults. We will investigate in which way the shadow concept can be made fault resilient by replication of the control structures.

Acknowledgements: Parts of the protocol have been implemented by M. Zepf. The comments of F. Hohl, M. Schwehm and M. Straßer improved the quality of the paper.

A The Protocol

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**(*place*, *„Hello“*) is sent to **place** and calls **receiveMessage**(*„Hello“*).

Methods bodies containing [here an example policy is presented] 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 Objects needed

<u>**Needed Objects**</u>

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:placeName;

Object ShadowProxy
Attribute agents:List of AgentProxy;

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

A.2 Methods in the Object Shadow

<u>**Shadow**</u>

```
timeToLive(from, agentId)
[here an example policy is presented]
    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 Basic Protocol with Proxies

<u>**Place: Methods**</u>

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

onArrival(agent)
    agentproxy = proxyList.find(agent.shadowId);
    if(agentproxy == null)
        agentproxy = new Proxy(agent.id, agent.timeToLive, agent.shadowHome, cur-
rentPlace);
        proxyList.add(agentproxy);
    else
        agentproxy.add(agent.agentId, agent.timeToLive);
    agent.proxy = agentproxy;
    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);
    else
        SendException (agent);

onTimer(agentproxy, agentId)            // time for the proxy path has ended
    agentproxy.remove(agentId);
    if(agentproxy.entries() == 0)
        proxyList.remove(agentproxy);
```

```
onTimer(agentproxy, agent)                  // allowance has not been sent in time
[here an example policy is presented]
    agentList.remove(agent);
    agentproxy.remove(agentId);
    if(agentproxy.entries() == 0)
        proxyList.remove(agentproxy);

onTimer(shadow, agentId)                    // 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.timeToLive     = creatingAgent.timeToLive;
    newAgent.timeOut        = creatingAgent.timeOut;
    newAgent.shadowId       = creatingAgent.shadowId;
    newAgent.shadowHome     = creatingAgent.shadowHome;
    onArrival(newAgent);
```

A.4 Finding Agents

<u>Place: Methods</u>

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

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

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

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

A.5 Mobile Shadows

Shadow: Additional Attributes

```
Object Shadow
    Method move(placeName);
    Attribute currentPlace:PlaceName;            // null if shadow is at home
    Attribute homePlace:PlaceName;
    Attribute timeToLive:Integer;
```

Shadow: Additional Methods

```
move(target)
    if(timeToLive != 0)
        sendShadow(target, this);
        if(currentPlace != null)                 // we are part of the path
            startTimer(timeToLive + timeOut, shadow);
        currentPlace = target;
terminateShadow()
    if (currentPlace != null)                    // the shadow has moved out
        sendTerminate(currentPlace, id);
    delete(this);
```

Place: 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.homePlace != place.name())    // only if not at home place
            shadow.timeToLive--;
            if (shadow.timeToLive == 0)
                sendCheck(shadow.homePlace, shadow.id);
                startTimer(shadow.timeOut, shadow);

onTimer(shadow, agentId)                         // agent has been killed
    shadow.remove(agentId);                      // due to timeout
    if (shadow.currentPlace != place.name() )
        sendRemoved(currentPlace, shadowId, agentId);

onTimer(shadow)
    shadowList.remove(shadow);                   // shadow path can be removed
```

```
receiveAllowance(shadowPlace, agentId, timeToLive)
    stopTimer(agentId);
    agent = agentList.findAgent(agentId);
    agent.timeToLive = timeToLive;
    agent.shadowHome = shadowPlace;
    proxyList.setTime(agentId, timeToLive);
receiveAllowance(shadowId, timeToLive)
    shadow = shadowList.find(shadowId);
    stopTimer(shadow);
    shadow.timeToLive = timeToLive;
receiveCheck(from, shadowId, agentId)
    stopTimer(agentId);
    if(currentPlace != place.name())                // not the current shadow
        sendCheck(currentPlace, 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, place.name(), agentId, timeToLive);
receiveCheck(from, shadowId)
    shadow = shadowList.find(shadowId);
    if(shadow != null)
        shadow.currentPlace = place;
        sendAllowance(from, shadowId, shadow.timeToLive);
receiveShadow(shadow)
    if(shadow.timeToLive != 0)
        if(shadow.homePlace != place.name())
            shadow.currentPlace = place.name();
            shadowList.add(shadow);
        else                    // shadow comes back home
            shadowList.find(shadow.shadowId);
            shadowList.remove(orig_shadow);
            shadowList.add(shadow);
            shadow.currentPlace = null;
receiveRemoved(shadowId, agentId)
    shadow = shadowList.find(shadowId);
    if(shadow != null)
        if(shadow.currentPlace != place.name())
            sendRemoved(currentPlace, shadowId, agentId);
        else
            shadow = shadowList.find(shadowId);
            shadow.remove(agentId);
```

B References

1. J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, M. Straßer. “Communication Concepts for Mobile Agent Systems”, in Mobile Agents ‘97, LNCS 1219, Springer-Verlag, pp. 123 - 135, 1997.
2. J. Baumann. „A Protocol for Orphan Detection and Termination in Mobile Agent Systems“, Tech. Report 1997/09, Fac. of Computer Science, U. of Stuttgart, 1997.
3. J. Baumann, F. Hohl, K. Rothermel, M. Straßer. „Mole - Concepts of a Mobile Agent System“, in WWW Journal, Special Issue on Software Agents, to appear.
4. J. Baumann, N. Radouniklis. „Agent Groups for Mobile Agent Systems“, in Distributed Applications and Interoperable Systems, H. König et al., Eds., Chapman & Hall, pp. 74 - 85, 1997.
5. 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, pp. 301 - 308, 1996.
6. General Magic, “Odyssey Web Site”. URL: http://www.genmagic.com/agents/
7. IBM. “The Aglets Workbench”. URL: http://www.trl.ibm.co.jp/aglets/
8. F. Mattern. “Verteilte Algorithmen”, Springer-Verlag, 1989.
9. G. Tel, F. Mattern. “The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes.“, ACM TOPLAS 15:1, pp. 1-35, 1993.
10. M. Shapiro, P. Dickman, D. Plainfossé. “SSP Chains: Robust, Distributed References supporting acyclic Garbage Collection”, Tech. Report No. 1799, INRIA, Rocquencourt, Frankreich, 1992.
11. M. Straßer, J. Baumann, F. Hohl. “Mole - A Java Based Mobile Agent System”, in Workshop Reader ECOOP ’96, d-punkt, pp. 327 - 334, 1996.
12. G. Tel. „Distributed Algorithms“, Cambridge University Press, 1994.
13. J. E. White. “Telescript Technology: The Foundation of the Electronic Marketplace”, General Magic, 1994.