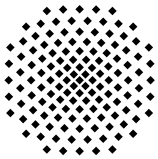


Universität Stuttgart
Fakultät Informatik

ATOMAS: A Transaction-oriented Open Multi Agent-System. Final Report

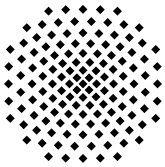
Authors:
Dipl.-Inform. M. Straßer
Dipl.-Inform. J. Baumann
Dipl.-Inform. F. Hohl
Dr. M. Schwehm
Prof. Dr. K. Rothermel

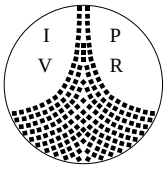
Institut für Parallele und Verteilte
Höchstleistungsrechner (IPVR)
Fakultät Informatik
Universität Stuttgart
Breitwiesenstr. 20 - 22
D-70565 Stuttgart

ATOMAS: A Transaction-oriented Open Multi Agent-System. Final Report

M. Straßer, J. Baumann, F. Hohl,
M. Schwehm, K. Rothermel

Bericht 1998/11
Juni 1998





ATOMAS:

A Transaction-oriented Open Multi Agent-System

Final Report

A Project Funded By Tandem Inc., Cupertino

22.6.1998

Authors:

Prof. Dr. K. Rothermel
Dr. M. Schwehm
Dipl.- Inform. J. Baumann
Dipl.- Inform. F. Hohl
Dipl.-Inform. M. Straßer

Institute for Parallel and Distributed
High Performance Systems
University of Stuttgart
Breitwiesenstraße 20-22
D-70565 Stuttgart

Contents

1 Introduction

The electronic marketplace of the future will consist of a large number of services located on an open, distributed and heterogeneous platform, which will be used by an even larger number of clients. Mobile Agent Systems are considered to be a precondition for the evolution of such an electronic market. They can provide a flexible infrastructure for this market, i.e. for the installation of new services by service agents as well as for the utilization of these services by client agents.

Mobile Agent Systems basically consist of a number of locations and agents (see Figure 1). Locations are (logical) abstractions for (physical) hosts in a computer network. The network of locations serves as a unique and homogeneous platform, while the underlying network of hosts may be heterogeneous and widely distributed. Locations therefore have to guarantee independence from the underlying hard- and software. To make the Mobile Agent System an open platform, the system furthermore has to guarantee security of hosts against malicious attacks.

(User) Agents are active, autonomous software objects, that reside (and are processed) on locations. They can communicate with other agents either locally inside one location or globally with agents on other locations. Mobile Agents furthermore can migrate from one location to another. Mechanisms for the communication between agents and for the migration of agents have to be provided by the Mobile Agent System.

Service Agents are interfaces to services. Next to the normal communication mechanisms between agents of the mobile agent system, they have access to services provided by the underlying host. Because of their machine dependent purpose, service agents are not mobile.

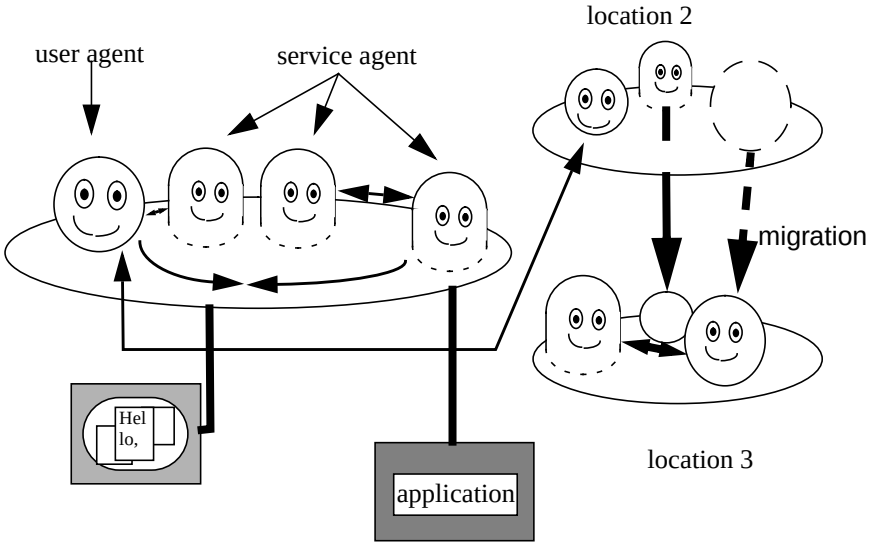


Figure 1.1. Mobile Agent System

The Atomas project aims in developing an open agent system as an enabling technology for the evolution of a electronic marketplace. This report documents the achieved results of the second year. Section 2 provides an overview over the objectives of the work packages for the first year and their completion state. Sections 3-??? present the results in detail.

2 Workplan and Project State

In the second year, the agent system architecure developed in the first year had to be extended to support migration and to provide reliable agent execution. Furthermore, requirements concerning security issues had to be investigated and the concepts developed during the project had to be evaluated. Therefore, four work packages had been identified (Section 2.1 - Section 2.4). In addition orphan detection for agents has been identified as a very important functionality to be supported by mobile agent systems and thus an additional work package (Section 2.5) has been added that concentrates on this aspect.

The following sections contain a short introduction into the work packages and the completion state of these work packages.

2.1 WP 2.1: Design and Implementation of Agent Migration

In work package 1.2, a special form of migration, called weak migration, had already been developed during the first year. This form of migration has the advantage that it can be implemented without changing the Java virtual machine and therefore allows to run the developed agent system on each architecture for which a Java virtual machine is available. Our experience in using this form of migration in several student thesises proved the validity of this concept. Therefore, no further work has been invested in this topic in the second year.

2.2 WP 2.2: Requirement Analysis Concerning Security

In the first year, driven by the importance of the security aspect of mobile agent systems, we already started this work package. Section 7 of the last years report investigates the security aspects of Mobile Agent Systems and identifies the problem areas which has to be handled.

In Section 3 of this report, a approach to partially solve one of the most difficult aspects of security of mobile agents systems,the problem of malicious hosts, is presented. This problem consists in the possibility of attacks against a mobile agent by the party that maintains an agent system node, a host.

2.3 WP 2.3 Recoverable Agents

An important prerequisite for the use of mobile agents in industrial environments is to provide reliable and fault tolerant execution of the agents. Section 4 describes two different approaches to provide the required reliability. In the first approach, fault-tolerant execution of agents on unreliable systems is provided. Section 4.1 summarizes two published papers on this topic. In the second approach, the reliability of agents is provided by using the TUXEDO platform on Tandem Himalaya to build the agent system.

2.4 WP 2.4 Developed Concepts and Implementation

In this work package, the developed concepts are summarized and implemented within the mobile agent system Mole 3.0. Next to the concepts for agent communication and agent migration

designed and implemeted in the previous year, version 3.0 of Mole now provides an extensive infrastructure for agents, including a resource manager, a directory service and a global naming scheme for agents. In order to support the design of agents, a graphical agent monitor allows to visualize the system behaviour as a whole or of a single agent in particular. Mole further provides a thread management unit to overcome some shortcomings of the Java virtual machine. Mole provides a simple means for installation and configuration of the system.

2.5 Orphan Detection for 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.

3 WP 2.2: Security

This section summarizes a paper published in [Vig98].

3.1 Abstract

In this report, an approach to partially solve one of the most difficult aspects of security of mobile agents systems is presented, the problem of malicious hosts. This problem consists in the possibility of attacks against a mobile agent by the party that maintains an agent system node, a host. The idea to solve this problem is to create a blackbox out of an original agent. A blackbox is an agent that performs the same work as the original agent, but is of a different structure. This difference allows to assume a certain agent protection time interval, during which it is impossible for an attacker to discover relevant data or to manipulate the execution of the agent. After that time interval the agent and some associated data get invalid and the agent cannot migrate or interact anymore, which prevents the exploitation of attacks after the protection interval.

3.2 Introduction

Mobile agent systems are expected to become a possible base platform for an electronic services framework (see e.g. [Mob96]), especially in the area of Electronic Commerce. In this application area, security is a crucial aspect since all parties involved require the confirmation that none of the other parties will break the rules without being punished. This requirement is not always fulfilled even in the traditional, non-electronic commerce. The anonymity of a worldwide communication network and the ease of automatic exploitation of security gaps in electronic applications make it necessary to meet this demand in the area of commercial transactions done by computers.

Mobile agents are entities that consist of code, data and control information (e.g. thread states). Mobile agent systems are platforms that allow mobile agents to migrate between different nodes of the agent system. From a more technical view, mobile agents can be compared to programs that migrate to nodes autonomously, while nodes offer the run-time environment of these programs including the program interpreters.

As in Mobile Code systems (e.g. the Java applet system), one aspect of security is the protection of the node, or *host*, against possible attacks of the mobile agent. Therefore, some of the security mechanisms developed in this field can also be applied to mobile agent systems. An example is sandbox security, i.e. the need of authorizing security-sensitive commands like the deletion of a file by a designated component. Other security mechanisms like authentication of single agents instances do not have a counterpart in mobile code systems and have to be designed using standard cryptographic techniques like encryption or digital signatures.

The reverse security issue, the protection of a mobile agent from possible attacks by a malicious host, is new as there are barely other areas where this aspect is important. Nevertheless, the protection of mobile agents from malicious hosts is — at least from the viewpoint of the owner of the agent — as important as the protection of the host from malicious agents. As we will see, apart from organisational solutions, no technical approaches to solve this problem without spe-

cial secure hardware exist so far. The solubility of this problem which is called the *problem of malicious hosts* is even estimated to be very low [FGS96].

This report presents an approach to solve most of the aspects of the problem of malicious hosts. This approach will cost both execution time and communication bandwidth and will require some time-critical restrictions, but gives the agent the possibility to do some security sensitive work without the danger of an immediate exploitation of sensitive data by the host.

3.3 The Problem of Malicious Hosts

The fact that the runtime environment (the host) may attack the program (the agent), plays hardly a role in existing computer systems. Normally, the party that maintains the hosts also employs the program. But in the area of open mobile agents systems, an agent is operated in most cases by another party, the agent owner. This environment leads to a problem, that is vital for the usage of mobile agents in open systems: the *problem of malicious hosts*. A malicious host can be defined in a general way as a party that is able execute an agent that belongs to another party and that tries to attack that agent in some way. The question of what action is considered to be an attack depends on the question which assurances an agent owner needs in order to use a mobile agent. If we try to achieve a protection level that is comparable to the one of agents that run on non-malicious, or *trusted* hosts, we can identify the following attacks:

1. spying out code
2. spying out data
3. spying out control flow
4. manipulation of code
5. manipulation of data
6. manipulation of control flow
7. incorrect execution of code
8. masquerading of the host
9. denial of execution
10. spying out interaction with other agents
11. manipulation of interaction with other agents
12. returning wrong results of system calls issued by the agent

To illustrate these attacks we will use a small purchase agent as an example. The purchase agent contains a data and a code block. Entries in the data block may include:

```
Address home = “PDA, sweet PDA”
Money wallet = 20$
float maximumprice = 20.00$
good flowers = 10 red roses
Address shoplist[] = empty list
int shoplistindex = 0
float bestprice = 20.00$
Address bestshop = empty
```

The central procedure `startAgent`, that is called by the host every time the agent arrives, could look like this:

```
1  public void startAgent() {
2
3    if (shoplist == null) {
4      shoplist = getTrader().
5       getProvidersOf(“BuyFlowers”);
6      go(shoplist[1]);
7      break;
8    }
9    if (shoplist[shoplistindex].
10     askprice(flowers) < bestprice) {
11      bestprice = shoplist[shoplistindex].
12                    askprice(flowers);
13      bestshop = shoplist[shoplistindex];
14    }
15    if (shoplistindex >= (shoplist.length - 1)) {
16      // remote buy
17      buy(bestshop,flowers,wallet);
18      // go home and deliver wallet
19      go(home);
20      if (location.getAddress() = home) {
21        location.put(wallet);
22      }
23    }
24    go(shoplist[++shoplistindex]);
25 }}
```

Using this example, the attacks listed above can be illustrated.

1. Spying out code

The code of the agent has to be readable by the host. Although this requirement can be restricted to the next instruction at a single point of time, this does not solve the problem since some hosts see almost all of the code because they execute most of the commands. In our example the host visited last executes nearly all the code. If the agent code is characteristic not only for a single, but a whole class of agents, the whole code of the agent may be known even before execution time. If an agent is generated out of standard building blocks (which is a good idea regarding code migration costs and ease of agent construction), the detail specification is available for building blocks like libraries or classes. Furthermore, these blocks can be explored by blackbox tests. Knowing the code leads to knowledge about the execution strategy of the agent, knowledge about the exact physical structure of code and data in the memory of the host and sometimes (by using data statements like initial variable assignments) to knowledge about parts of the agent data.

2. Spying out data

The threat of a host reading the private data of an agent is very severe as it leaves no trace that could be detected. This is not necessarily true for the consequences of this knowledge, but they can occur a long time after the visit of the agent on the malicious host. This is a special problem for data classes such as secret keys or electronic cash, where the simple knowledge of the data results in loss of privacy or money. In our example, the money variable would be security sen-

sitive when it is represented in a way that the binary number of the “coin” **is** the money and therefore can be used as real world cash. But there are also other classes of data, which can be used for an attack although they have not the nature of classes like e-cash. In our example, the knowledge of the maximum price or the best price so far can be used by a malicious host to offer flowers for a slightly lower amount than the competitors, although the regular price is much lower.

3. Spying out control flow

As soon as the host knows the entire code of the agent and its data, it can determine the next execution step at any time. Even if we could protect the used data somehow, it is rather difficult to protect the information about the actual control flow. This is a problem, because together with the knowledge of the code, a malicious host can deduce more information about the state of the agent. In our example, we can recognize whether an offer is better or worse than the best offer so far by simply watching the control flow, even if we could not read any data.

4. Manipulation of code

If the host is able to read the code and if it has access to the code memory, it can normally modify the program of an agent. It could exploit this either by altering the code permanently, thus implanting a virus, worm or trojan horse. It could also temporarily alter the behaviour of the agent on that particular host only. The advantage of the latter approach consists in the fact, that the host to which the agent migrates cannot detect a manipulation of the code since it is not modified. Applied to our example, a malicious host could modify the code of the agent with the effect that it prefers the offer of a certain flower provider, regardless of the price.

5. Manipulation of data

If the host knows the physical location of the data in the memory and the semantics of the single data elements, it can modify data as well. In our example, the host could cut down the shop list after setting the offer of the local flower provider as the best offer.

6. Manipulation of control flow

Even if the host does not have access to the data of the agent, it can conduct the behaviour of the agent by manipulating the control flow. In our example, the host could simply alter the flow at the second or third if statement, forcing the agent to choose the offer of the shop preferred by the host as the best.

7. Incorrect execution of code

Without changing the code or the flow of control, a host may also alter the way it executes the code of an agent, resulting in the same effects as above.

8. Masquerade

It is the liability of a host that sends an agent to a receiver host to ensure the identity of that receiver. Still, a third party may intercept or copy an agent transfer and start the agent by masking itself as the correct receiver host. A masquerade will probably be followed by other attacks like read attacks.

9. Denial of execution

As the agent is executed by the host, i.e. passive, the host can simply not execute the agent. This

can be used as an attack e.g. in the case that a host knows about a time limited special offer of another host. The host simply can prevent the detection of this offer by the agent by delaying its execution until the offer expires.

10. Spying out interaction with other agents

The agent may buy the flowers remotely from a shop situated on another host. If the interaction between agent and the remote flower shop is not protected, the host of the agent is able to watch the buy interaction even in case the host cannot watch the execution of the agent. In our example, the host could read e.g. `wallet` and spend the stored money.

11. Manipulation of interaction with other agents

If the host can also manipulate the interaction of the agent it can act with the identity of the agent or mask itself as the partner of the agent. In our example the host can e.g. redirect the buying interaction to another shop, or it can interrupt the interaction e.g. to prevent spending the money by the agent.

12. Returning wrong results of system calls issued by the agent

In line 20 of the example code ("`if (location.getAddress() = home)`"), the agent requests the name of the current location. Here the host could mask itself as the agent's home location by returning the corresponding address. The agent then thinks that it is at home and delivers the wallet to the host.

After stating the problem we will now have a look on possible solutions. First we will examine some approaches that try to prevent single attacks. In the next section we will see an approach that try to restore the autonomy of the agent, the so called *blackbox approach*.

3.4 Existing Approaches

As mentioned above, a malicious host is defined as a party that is able execute an agent that belongs to another party and that tries to attack that agent in some way. This also means that malicious hosts are only a problem for agents that cannot trust a host in advance. In this case *trust* means, that the owner either knows or hopes that the operator will not attack. Therefore, some approaches (see e.g. [FGS96]) exist that try to circumvent the problem of potentially malicious hosts by not allowing agents to move to non-trusted hosts. There are also approaches that use a trust approach to protect hosts from agents by not allowing to accept agents that have been on non-trusted hosts before. The problem of these approaches are that trust in this context is absolute (you do not hide anything from a trusted node), and that it is not always clear in advance whether a host is trusted or not. This can severely reduce the number of hosts an agent might migrate to. Even if an owner trusts a big company when it comes e.g. to accounting, it may not want them to see its secret communication key. If an agent has to obtain prices for a flight, it cannot trust the host of an air line or any other host that is maintained by a company related to an air line and so forth.

Another "trust" approach is the *organizational* solution: the agent system is not open in the sense that everybody can open a host, but only trustworthy parties can operate hosts. This is the approach General Magic [GM96] used for its agent system application, e.g. PersonaLink[1], that was operated by AT&T [Mob96].

As trust is a relationship between agent and host which often cannot be determined in advance, a commonly used notion of trust, *reputation,* is used in another approach [RJ96]. This also is problematic, as we have seen that trust depends on the task an agent has to fulfill. A reputation approach, where betrayed agents can complain about malicious hosts, that in turn, lose reputation, can also result in a new security problem. Agents could attack hosts using a "character assassination" attack, by simply complaining about being betrayed.

Another approach [Vig97] enables an agent to *detect and prove modification attacks* in order to allow the owner to use legal or organizational ways to get its damage refunded. But this approach cannot prevent other attacks, and it assumes an organizational or legal framework for an agent system. In the first case, such an organizational framework may not exist in an open agent system without a central organization. In the second case it seems to be not realistic to assume such a legal framework on an international level, since also other laws required by new technologies, e.g. for data protection and privacy, are far from being homogeneous or even widespread.

Since the problem is the wrong behaviour of the executing environment, in contrary to a behaviour that meets the specification, another class of approaches (e.g. [Pal94]) uses specialized, attack-proven *hardware* that can ensure its integrity. These approaches therefore require the usage of this hardware in every host, which is currently a too restricting assumption.

As the presented approaches either do not protect from all the attacks, or do not allow open mobile agent systems, a more adequate approach is needed.

3.5 Blackbox Security: The Idea

In this section we will discuss an approach that is able to protect an agent from most of the attacks mentioned in the first section. The central idea of this approach is to generate an executable agent from a given agent specification which cannot be attacked by read or manipulation attacks. This agent is considered to be a "blackbox", if the following applies:

Def: Blackbox Property

- an agent is a blackbox if:
  1. at any time
  2. code and data of the agent specification cannot be read
  3. code and data of the agent specification cannot be modified

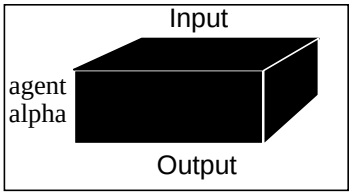


Figure 3.1. Blackbox property

If this definition can be applied to an agent, only input to and output from the blackbox can be observed.

The "conversion mechanism" that generates an agent with the blackbox property uses configuration parameters that allow to create different blackboxes out of the same specification (see Figure 3.2). These parameters allow to prevent dictionary attacks. Dictionary attacks guess the attributes of the blackbox by converting a number of agent specifications and compare the cre-

1. PersonaLink was a service that allowed users to send electronic mails that carried agents. It was based on the Telescript mobile agent system.

ated blackboxes with the attacked one.

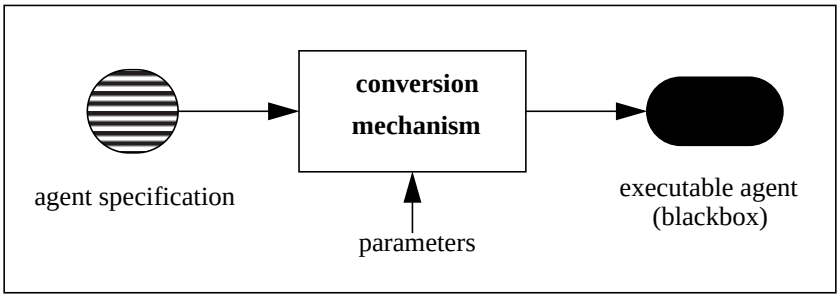


Figure 3.2 Blackbox approach

If an agent fulfills the blackbox property defined above, it is autonomous in the sense that if a hosts executes that agent, the host cannot interfere with this execution in a directed way. If an agent reaches that level of autonomy, it can be protected from other attacks. Masking of the host or reading and manipulating the interaction of the agent with other parties can then be prevented by using conventional mechanisms from the area of stationary distributed systems.

The problem now is to ensure the blackbox property. Currently, there is no known algorithm to fully provide blackbox protection even if one other approach exists that seems to proceed in this direction. It is called Mobile Cryptography.

3.6 Mobile Cryptography

This approach does not call itself a blackbox approach, but it can be classified in this category. Sander and Tschudin describe in [ST98a] and [ST98b] a way to use *encrypted programs* as a means to protect agents from malicious hosts. Encrypted programs are programs that consist of operations that work on encrypted data. Agents are produced by converting a agent specification into some executable code plus initial, encrypted data. Since the attacker cannot break the encryption of the data, it cannot read or manipulate the original data. See [ST98a] for a detailed description of the Mobile Cryptography approach.

The advantages of this approach over the one that will be presented in the next section are:
- the protection of the agent is easily provable
- the costs of the protection are probably small
- the protection is not time-limited

The current restrictions of the Mobile Cryptography approach are:
- random programs cannot be used as the input specification; currently only polynomial and rational functions can be used for this purpose
- the interaction model of the agent suffers the restriction that cleartext data can be sent only to trusted hosts

The extension of the approach to recursive functions and Turing machine program equivalent mechanisms are subject to future work. As soon as the latter can be used as an input to the conversion function, encrypted programs have also the blackbox property. However, even now most of the aspects described in this article, which do not rely on the specific conversion mechanism, apply also for encrypted programs.

The second restriction (cleartext data can be sent only to trusted hosts) is not mentioned explicitly. Still, receivers can only read encrypted output of the agent when they know the decryption function (which includes a potential key). If an attacker is able to decrypt the output of an protected agent, it is likely that it can also attack the agent itself.

3.7 Time Limited Blackbox Protection

As we have seen, the only known approach that tries to provide fully blackbox protection is currently not applicable to every existing agent. In order to remove this restriction, we redefine the blackbox property definition in a way which differs in the statement about how long the blackbox property is valid. Now we do not assume that the protection holds forever, but only for a limited, known minimal time interval known in advance. Therefore the definition is now:

Def: Time Limited Blackbox Property

- an agent is a blackbox if:
  1. *for a certain known time interval*
  2. code and data of the agent specification cannot be read
  3. code and data of the agent specification cannot be modified
- *attacks after the protection interval are possible*
  4. *but these attacks do not have effects*

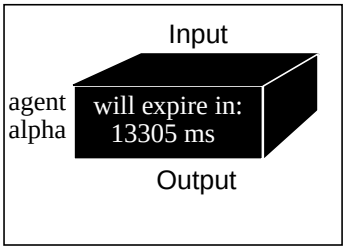


Figure 3.3 Time l. blackbox property

To make the protection time interval explicit, an *expiration date* is attached to the blackbox.

Although this definition is weaker than the original blackbox property and results, as we will see, in more complex mechanisms, it has one big advantage: there is a way to achieve this. Before this way is sketched, we examine what changes if blackboxes are time-limited.

3.8 What Is Changing If the Blackbox Is Time Limited?

For achieving the requirement that attacks after the protection interval do not have effects, we have to examine the circumstances under which time limitedness affects processing. To do this, four different interaction scenarios are introduced. It will be argued that effects of an attack can only occur when information of the agent is communicated to third parties.

3.8.1 No communication with a third party

In this scenario neither the agent nor the host communicates with a third party. Although this is a merely academic setting, it demonstrates that the temporal aspect is of no importance in this

context. Even if the host successfully attack the agent, nothing results from these attacks.

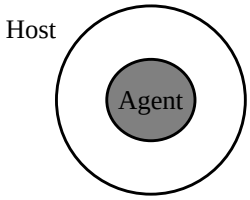


Figure 3.4 No communication

3.8.2 Communication only with trusted servers

Here the agent communicates only with a *trusted third party*. A party can be considered as trusted if this party never attacks. These two partners can establish a secure communication channel to prevent attacks by the host. Time limitedness of the agent plays a role in this scenario since the communication partner has to know whether it can still trust the agent or not. If the host would have been able to attack the agent, the attacker could use the agent to mask itself as the agent. Since attacks can only take place after the protection interval, the trusted server has to know the expiration date associated with the agent before it starts communication. This can be done using an extended key certificate (see Figure 3.6). The resulting overhead is acceptable since secure communication requires already authentication of the partners

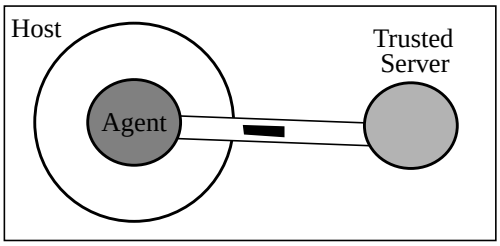


Figure 3.5 Communication only with trusted servers

Figure 3.6 Extended key certificate

3.8.3 Communication with untrusted servers

In this scenario the agent communicates with either an untrusted third party or with the host, which is by definition untrusted (see Figure 3.7).

We have to distinguish two kinds of data that can be communicated: token and non-token data.

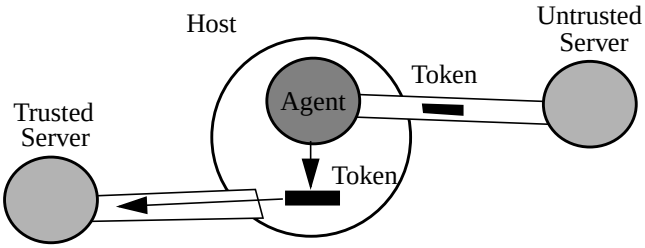


Figure 3.7 Communication with untrusted servers

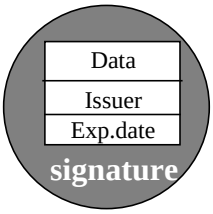


Figure 3.8 Token structure

Token data are self-contained documents that depend on the identity of the issuer. Therefore they often bear digital signatures. Examples for tokens are electronic money coins, secret keys and capabilities. The problem with tokens is, that an attacker may use or trade them without having obtained them regularly. Therefore, also tokens have to bear expiration dates to prevent the usage of tokens that could have been obtained by attacking the agent. Every party that receives a token by another party thus has to check whether the expiration date of the token has passed or not. To do that, this party has to be able to get the correct global time. This means that time limitedness always require synchronized clocks. Note that it is not necessary for the party that sends a token to know the current time. Only the party that issues a token and the party that receives a token have to have this information. The issuer needs it to add the protection interval to it. The receiver needs the current time since if a party accepts an outdated token, no other party will accept it in return. The drawback of the expiration date is, that a token cannot be protected after the expiration date. Thus, tokens which need a larger protection interval must not be transported by the agent. This can be the case for some existing token systems that do not include expiration aspects or which cannot be extended by this aspect. A good example for tokens that cannot be protected in agents are secret keys of an agent owner since they are valid normally for a long time.

Non-token data is everything else. Examples for this category are simple values that do not need to be protected and values that are security sensitive like the maximum price. The black-box property guarantees that they cannot be read or modified before the expiration date has passed. They cannot be used against the agent or its owner since they not depend on the identity of the issuer. Since non-token data cannot be used to interact with third parties, it does not need to be protected against modification attacks after the protection interval. Although nothing has to be done to protect non-token data, there is a restriction for these elements: an agent must not transport non-token data, that can be used to attack the owner of the agent and whose protection interval has to be larger than the lifetime of the agent. An example for such data could be a variable describing the maximum price for a good that is valid for all purchasing agents of a user ever used. Fortunately, data elements with a larger protection need do not seem to occur very frequently in reality.

Note that this scenario does include both planned interaction of the agent with an untrusted party or unplanned interaction by an attack of the host. Since an “unplanned interaction”, i.e. a read attack by the host can only take place after the expiration date, all allowed tokens are also

outdated then and non-token data does not have to be protected any longer due to the mentioned restriction.

3.8.4 Migration of the agent

The last scenario comprises the remaining possibility to explicitly communicate agent information: the migration of the whole agent to a new host. The problem here is, that the agent may have been “overtaken” or tampered by the host after the expiration date. Although it is unlikely that the code of the agent was manipulated by the host since it is rather easy to protect constant code from manipulation attacks by using digital signature techniques, an attacker could have been altered values that are not protected by the signature, i.e. mainly variable data.

Therefore, the receiving host has to ensure that the arriving agent is still valid, i.e. that its expiration date has not passed already. As we have seen the agent will probably be protected by a signature, and all we have to do is to either include the expiration date into the constant part of the agent allowing the signature to also protect this date or to use the extended key certificates we introduced above. The receiver then can simply check the signature of the agent and the validity of the agent by checking the expiration date.

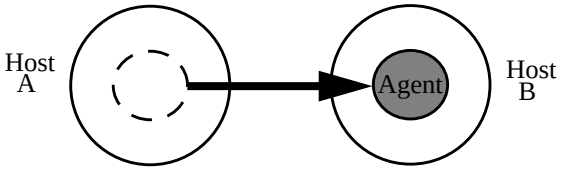


Figure 3.9 Agent migration

As we have seen, it is possible to compensate most of the effects that occur when agents are subject of time-limitedness. The next question to answer is how such a protection can be reached.

3.9 How Can We Reach Time Limited Blackbox Protection?

The lack of approaches that protect agents from host attacks is based on the observation that a host is always able to read every bit of the memory and the content of every variable and to know the memory location of every line of code. Therefore, some authors conclude, it is impossible to prevent e.g. read attacks.

While this observation is always true for the “semantics in the small”, i.e. the meaning of these elements for the next execution step, it is not necessarily true for the “semantics in the large”, i.e. the meaning of these elements to the overall semantics of the application. An example for this difference is the code in Figure 3.10 where you can of course put the finger on every statement and every variable, but to explain the meaning of a statement or a variable in relation to the overall result, you have to think about it (the code fragment computes the difference of two two-bytes-numbers).

```
w[6] = b[3] - b[5];
w[7] = b[2] * 256;
w[8] = w[7] + w[6];
w[5] = w[8] - b[4] * 256;
b[0] = w[5] DIV 256;
b[1] = w[5] MOD 256;
```

Figure 3.10 A code fragment

This effect results from the fact, that this overall semantics is not expressed by code, but by the "mental model" of the programmer or the reader of a program. To attack an agent, the human attacker has to have such a mental model of the code in order to find certain points in the code or values that are interesting for the attacker.

The central idea now is not to allow an attacker to build such a mental model of the agent in advance, i.e. before the agent arrives, and to make the process of building this model a time-consuming task. The first goal is reached by creating a new "form" of the agent dynamically, in an unpredictable, manner at the start of the protection interval. The second goal is reached by using conversion algorithms that produce a new form that is *hard* to analyse. In this context hard means that the analysis should take as much as time as possible. These conversion algorithms are therefore called *obfuscating* or *mess-up algorithms*. Note that the approach does not assume that it is impossible for the attacker to analyse the agent, the analysis simply takes time. The assumption is that a lower bound of this time can be determined and that this time interval is large enough for most agent applications on one host.

3.9.1 Agent mess-up algorithms

The task of a mess-up algorithm is to generate a new agent out of an original agent, which differs in code and data representation but yields the same results. This means, that the specification of the agent is given as an executable, unprotected agent. Agents consists of executable code and some data. To prevent dictionary attacks (see Section 2), it has to use a random parameter that allows the algorithm to create different new agents out of a single original one (see Figure 3.11).

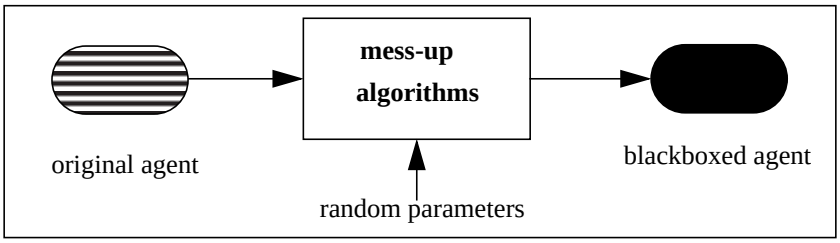


Figure 3.11 Time-limited blackbox approach

To achieve the requirement that a blackbox protected agent that is hard to analyse, the designer of a mess-up algorithm has to take into account two key aspects: the attributes of an agent that can be modified and the abilities and characteristics of the attacker

3.9.2 Agent attributes that can be modified

Statements

A statement has a type and a location in a program (it also consists of data, but this aspect is viewed below). The *type of a statement* can be hidden until the statement is executed by dynamically creating it at runtime. This is possible by using e.g. self-modifiable code. The *location of a statement* can also be hidden, either implicitly by using dynamic code creation or explicitly by hiding a certain statement into other statements.

Data

Data, i.e. variables and constants, consist of a type, a value and a location. The t*ype of a data element* can be hidden until the data is needed. This is even normal for languages that use dynamic typing as e.g. Smalltalk. The *value of a data element* can be hidden. One way to achieve this is to replace element accesses by accesses on subelements and to translate operations on the data elements by operations on the subelements. This results in an execution where the value of an data element never occurs as a whole. Finally, the *location of a data element* can be hidden either statically, e.g. by splitting up the element and distributing the parts, or dynamically by e.g. allowing the element to move around in the data area.

3.9.3 Abilities and characteristics of the attacker

To model the properties of the attacker, we have to distinguish two cases.

In the first case, the attacker does not know the original version of the agent in advance. Therefore, a human has to analyse the blackbox to build up a mental model. Although it can use the aid of computerized tools to do this, humans tend to be far too slow compared to the execution speed of computer. This slowness cannot be reduced fundamentally since it is not possible to speed up humans. Therefore, the next case seems to be much more relevant.

In the second case, the attacker does know the exact specification of the agent in advance. This case is probably the common one if most agents in an agent system are instances of a set of standard agents. If it is possible to identify the type of an agent, i.e. the original agent, then the exact specification is accessible. If now an attacker knows the exact specification, it can automate the attack by generating a program that tries to compute only a few or even a single attribute of the agent, e.g. the current location of a certain variable in the blackboxed agent. In this case the attack can be accelerated by using faster computers or by employing several computers in parallel.

In both cases the generated code has to be constructed in a way that standard program analysing techniques such as program slicing, data flow analysis or program abstraction, cannot be used to analyse the agent before the expiration date has passed. Good mess-up algorithms do not allow the complete analysis to be done statically, but also require to run the agent at least partially.

Let us now have a look at three example algorithms.

3.9.4 Examples for mess-up algorithms

The most important aspects are the structure and attributes of the used mess-up algorithms, as they decide about the protection strength of the security mechanism. Therefore, three mess-up algorithms will be sketched here.

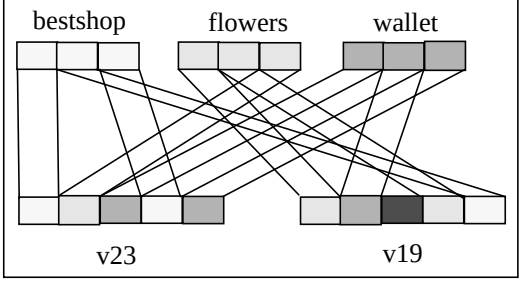
Variable Recomposition

This algorithm takes the set of program variables, cuts each variable content into segments and creates new variables that contain a recomposition of the original segments. The original variable accesses in the program code are then adapted correspondingly. In Figure 3.12a, you can see the original variable access, Figure 3.12b defines a scheme for recomposing two new variables v23 and v19 from the contents of three original variables.

Figure 3.12a Original variable access

```
5  buy(bestshop,flowers,wallet)
6  go(home)
```

Figure 3.12b Variable recomposition



The access code for the new variables as displayed in Figure 3.12d can therefore be created automatically, given the recomposition scheme, by using conversion functions (see Figure 3.12c) that create the original values from the new variables. As a result, now there is no direct relationship between variables and processing model elements like the maximum price from our example. The variable names are now meaningless and the data representation is rather complicated.

Figure 3.12c Conversion functions

```
public Address c7(Bitstring b)
public Good c4(Bitstring b)
public Money c3(Bitstring b)
public Address c34(Bitstring b)
```

Figure 3.12d New variable access

```
5  buy(c7(v23[0]+v19[4]+v23[3])
      ,c4(v19[0]+v19[3]+v23[1]),
      c3(v23[2]+v19[1]+v23[4]))
6  go(c34(v21[4]+v19[2]+v21[2]))
```

Conversion of Control Flow Elements into Value-Dependent Jumps

The next presented mechanism is a *conversion of compile-time control flow elements into run-time data dependent jumps.* Control flow elements like `if` and `while` statements allow the programmer to imagine the potential control flow even at compile time as these statements make control flow explicit. If we convert these elements into a form that depends on the content of variables, the control flow cannot be determined as easily as before. This dependence can be achieved by the usage of jumps that are bound to variable contents, e.g. switch-statements. The effect can even be strengthened by using complex variable expressions instead of using simple variables.

Figure 3.13a Original code

```
if (a(b) < c) {
   b = s(d(e) + f);
}
```

Figure 3.13b Converted Code

```
z= 0
DO
   if (z=0) then t1 = a(b); z=1; continue;
   if (z=1) then t2 = t1 < c; z=2; continue;
   if (t2) then t3 = d(e); z=3; continue;
   if (z=3) then t4 = t3 + f; z=4; continue;
   if (z=4) then b = t4; z=5; continue;
   if (z=5) then break;
LOOP
```

Deposited Keys

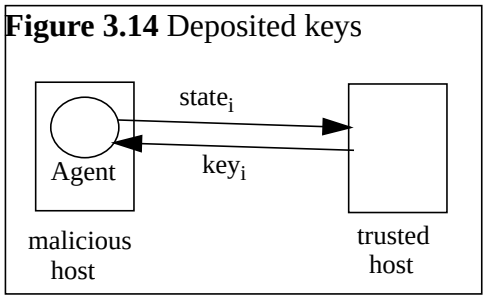


Figure 3.14 Deposited keys

If the whole protection information is included in the agent, an attacker is able to break that protection sooner or later. If we can encrypt parts of the agent or identify other information, that is both small and important for the execution of the agent, we can "externalize" this information on another, trusted server. The idea is to let the agent request these information parts, or keys, from the trusted server by indicating the state of the agent. An example for one type of keys can be found in Figure 3.13b where the numbers printed in bold denote data that "interconnects" the statements. If these numbers are not present in the code, the attacker is not able to analyse the agent and, therefore, to attack the agent before runtime. The trusted host will deliver them to the agent when the right state is indicated.

There are, of course, more and better algorithms, but the above examples demonstrate some of the principles they have to follow:

- the algorithm needs to be parametrizable with a very large parameter space in order to avoid dictionary attacks
- it must not be possible to break the protection without running the code
- it may be useful to take out parts of agent and to put these parts on trusted nodes

Each single of these algorithms may be not that strong if they would be used alone. It can be expected that a *combination* of these algorithms is much stronger than the sum of the single strengths. Therefore, our approach uses a "chain" of mess-up algorithms. To illustrate the effects of the mess-up algorithms, we want now to sketch possible algorithms that try to break the protection generated by the example algorithms.

3.9.5 Counter attacks

Having the three example mess-up algorithms in mind, we want now have a look at possible *counter attacks*. Counter attacks are algorithms that try to break the protection, i.e. mess-up algorithms.

Variable Recomposition

There are at least two possible approaches: we can try to guess the variable layout by analysing the access to a byte statistically over the operations known to access certain original variables. The other approach tries to read the original variables simply by reading the parameters of the calls of the known procedures. This means in our example, that the attacker sees the values of `bestshop`, `flowers` and `wallet` as soon as `buy` is called as these are the contents of the parameters of this procedure.

Both attacks assume knowledge about the original procedures. Fortunately, this attack is not that important as either the known procedure is a system call of the host or a call of an "internal" procedure. In the first case, the parameters do not have to be protected as they are by definition not secret (they are delivered to the host). In the second case we can dissolve the internal procedures into code of the main procedure, so that they are not visible any more.

Conversion of Control Flow Elements into Value-dependent Jumps

The presented version of this algorithm is rather easy to break as it can be analysed without having to run the code. All we have to do is to create the original statement out of the if-statements. The computational complexity of this is roughly proportional to the number of if-statements, which corresponds at most to the number of single language expression nodes of the syntax tree. We can prevent such an approach by replacing the constant numbers in the if-conditions by more complex expressions and by adding another algorithm that adds more dynamics to the computation of the conditions, e.g. the Deposited Keys mechanism.

Deposited Keys

We can attack this algorithm by creating every possible state of the agent and by requesting all the keys that are associated to these states. We then have all the runtime informations of the agent and can try to analyse it. The question is, how the attacker gains all the states of an agent. If the states can be associated to the execution of the agent (e.g. by computing a key that has to be delivered with the request), the host has to execute the agent. We then can control the attack by the trusted host since it can notice the execution of the agent.

3.9.6 Problems with mess-up algorithms

The first main problem is that the protection intervals have to be of a “useful” length. Useful in this context means the question of how long a protection interval has to be in order to allow the agent to do something useful. The answer depends of course on the task the agent has to fulfill on a host, but with an interval that allows two “long-range” migrations, some execution time and enough time for the protection overhead, most applications should be in range. If the protection interval is longer, the agent can migrate to more hosts or compute a longer time on every host. If the protection interval is much smaller, the possible application areas of the protected agent is severely restricted.

The second and even bigger problem is the question of how to determine these protection intervals from the used mess-up mechanism. Here, the usage of cryptography to protect data has a valuable advantage: it is possible to express the protection strength of the crypto algorithm in terms of the needed computational power. This is possible because there are known algorithms that are able to break the encryption. Sometimes (as with RSA), these algorithms are not necessarily the best possible mechanisms, but the best known, even after some decades of research. More often, the complexity class of the problem that breaks the encryption is known to be too hard to be computed, even in case of technological progress if the key is long enough.

Compared to cryptography, blackbox protection has two advantages:

- there is no receiver that has to apply the reverse encryption process
- the identity (and - in limits - the specification) and the order of the used algorithms does not have to be known in advance

We could now try to apply the same mechanism of determining the protection strength of crypto algorithms to mess-up algorithms. Unfortunately, this approach seems to be very difficult. The reason for that is the current lack of a formal model of the agent mess-up and the associated counter algorithms. While in traditional cryptography the problem of breaking an

encryption can be tracked down to a well-defined mathematical problem, the possible attacks against a blackbox protected agent are numerous, and different in nature. Future research has to develop a model that expresses e.g. the hiding of the location of a variable on several places and that computes the complexity to find that location. Fortunately we do not have to formalize the process of building up a mental model of a program by a human as we have excluded this possibility due to the lack of attack performance of humans in Section 8.3.

However, the current lack of a formal model is not an immanent problem of the approach. It is an open problem that has to be solved in order to both estimate the strength of the protection and to compute the current protection interval for a specific agent. This computation will then take into account the average computational need for solving the problem of breaking the blackbox protection and estimate how much computational power an attacker will stake. Since the computation will be done before an agent migrates the first time, the estimation can be adjusted according to the existing technology.

3.9.7 How can a blackbox protected agent be created?

To create a protected agent, token type data has to be converted into tokens that bear an expiration date and that are signed digitally. In the next step all security sensitive library calls (like calls of an encryption function) have to be replaced by the corresponding library code. Afterwards the code mess-up algorithms are applied to the code and the data of the agent. Finally, the agent has to be signed digitally after receiving the agent expiration date. Now the agent is ready to migrate.

3.9.8 Recharging of protected agents

If the “maximum distance” of the agent is determined by its expiration date, is it possible to “recharge” the agent in order to allow it to migrate further? Due to the nature of the code mess-up algorithms, any host could convert the agent to a new form without having to know its internal structure or the contents of the original data. Therefore, we could assign this task to any host that does not cooperate with any malicious host the agent has visited or will visit. Unfortunately, the expiration dates of the agent and of the transported tokens are a problem as they cannot be modified that easily. The first problem is that the agent has to be assigned with a new expiration date and signed digitally by a party that the agent (or its owner) trusts. This also incurs, that the agent gets a new identity, as it differs at least in the expiration date. The second problem is, that the tokens have to be replaced by new ones. If you think of electronic money coins, you have to change them into new coins with a new expiration date, while tokens that have no real value like keys, can be easily created. All of this can only be done by a trusted host. If the agent has checked the identity of the trusted host, it delivers the tokens that have to be replaced and gets the new ones in return. An alternative to recharging an agent is to extract the state of a nearly expired agent and to “inject” it into a new agent “hull”, thus creating a new agent that contains the state of the old one. The advantage of this alternative is that it prevents the delay that would be needed to mess-up the old agent after its arrival.

Now we have seen how to achieve time limited blackbox protection. But what can we do to prevent other attacks by the host?

3.10 Which Other Attacks by Malicious Hosts Can Be Prevented Using Blackbox Protection?

Even if an agent is protected by time-limited blackbox security, there are still some possible attacks:

- a malicious host can try to mask itself as another, perhaps trusted host
- a malicious can try to read and manipulate the interaction of a hosted agent with a third party and
- a malicious can return wrong results when the agent is calling system library procedures

While there is no known protection from the latter attack apart from verifying the answer by another third party (but then using library code seems to be rather redundant), the first two attacks can be prevented.

This is possible since a blackbox protected agent is autonomous again, i.e. that if a hosts executes that agent, the host cannot interfere in this execution. This allows us to use the same mechanisms to prevent the mentioned attacks as in distributed systems where the parties reside on different, and therefore autonomous, nodes.

We can prevent masking of hosts by using existing authentication methods using symmetric or asymmetric encryption schemes. We can even strip down these protocols a little bit since no third party can read the local communication between the agent and the host.

We can also prevent attacks against the interaction of an agent with another party by using secure channels between the interaction partners. These channels are obtained by exchanging session keys between the partners and by encrypting the traffic between them. Since in this scenario, the malicious host can be modeled as an attacker on a connection between two autonomous nodes, the protocols do not have to be modified.

3.11 New Attacks: Sabotage and Blackbox Testing

If there is an agent protection scheme like the one described in this report, one can imagine attacks that rely on the characteristics of this scheme. One attack is *sabotage*, or the action of destroying parts of the agent without being detected. As an agent contains data that might change during execution, the attacker can simply modify single bits of the data area without knowing about the effects to the agent. Fortunately, this attack is very similar to the problem of data that is sent over an insecure network. Therefore, similar error detection or even correction mechanisms like CRC, computed by the agent itself, can be used as long as the attacker cannot detect the detail structure of the mechanism. It is easy to circumvent a CRC algorithm if the exact mechanism is known and if it the borders of the protected data elements can be seen.

Another attack is the *blackbox test*. Its aim is to determine characteristics of the inside of the “black box” by executing the box with different input parameters and by watching the effects. The recorded reactions can be formal results like output values or characteristic “activity patterns”. In our example, the attacker could execute the agent until it tries to buy the flowers, starting over and over with the initial agent. The only value that is changed over the tests is the price for the flowers. When the agent finally wants to buy, the attacker knows the price that is both the lowest so far and that is below the maximum price. Even if the agent would not buy

the flowers immediately (it might want to ask at least three different providers), the attacker can watch whether the data of the agent changes. If this is the case, it is very likely, that this agent has memorized a better price. If it comes to countermeasures, two goals have to be reached: first, the parallel execution of the same agent has to be suppressed, e.g. by using a trusted third party that is informed by the agent about its execution. Second, the very fast execution of an agent has to be prevented, e.g. by using a similar interaction with a trusted host. Finally, activity patterns can be covered up by inserting and executing dummy code.

3.12 What Blackbox Security Costs

Protecting agents using blackbox security is not for free. Since the costs mainly depend on the class of an agent, it has to be decided per class whether this kind of protection is appropriate or whether the agent should operate immobile from a trusted host via remote communication. For calculating the costs, we can distinguish four classes of costs that result from blackbox security:

costs at creation time

These are the costs for converting the original agent into the new form. These costs are not important for the execution time of the agent, only for the “delay” of starting the execution. If we get an agent with low execution time overhead, we can accept higher creation time costs.

costs at transmission time

This is the size overhead of the agent, since the transmission time is determined by the size. The main problem here is the fact, that agents have to transport all library code that is security sensitive instead of using the corresponding system library at the target host. An example are the J/Crypto libraries from Baltimore Technologies that implement cryptographic functions like DES, RSA, SHA-1 and MD-5 and which consist of 200 KB of Java bytecode.

costs at execution time

The execution time overhead results on the one hand from the computations that are introduced by the mess-up algorithms and on the other hand from the execution time of the transported libraries if this time is longer than the execution time of a system library call. There are also costs if communication with remote trusted nodes is needed (e.g. in the case of Deposited Keys).

“costs” by not using efficiency enhancing mechanisms

Due to the blackbox mechanism, it is possible, that mechanisms enhancing efficiency cannot be used by protected agents. One example is the fact, that blackbox agents are not modular and hence cannot use code caching mechanisms as the code is different for every agent even if providing exactly the same functionality.

3.13 Conclusions and Future Work

Blackbox security is a new approach to solve the problem of malicious hosts, a problem in the area of mobile agent security, that has been rated as not solvable by software means. The pre-

sented approach does not prevent every possible attack. It is still possible for the host to deny the execution and to return wrong system call results to the agent. It is further still possible to read and to manipulate data and code, but as the attacker cannot determine the role of these elements for the application, the attack results are random. The approach is able to guarantee a certain protection time interval. Therefore, the agent and its transported data get invalid after this “expiration date”. For the purpose of comparing the expiration dates with the current time, synchronized clocks are necessary. As the strength of blackbox security depends on algorithms that “mess-up” code and data of the agent, these algorithms have to be constructed in a way that can guarantee the protection time interval, which also have to be of a useful length. As we have seen, this kind of security is not for free, but costs both in terms of execution and transmission speed. We expect therefore, that blackbox security will be applied only to agents that transport money-like values or security sensitive data such as secret keys.

We will implement a framework for blackbox security for our own Java-based agent system, Mole [Mole]. At the moment, no overall implementation of the approach exists as it is a complex framework that needs a lot of modifications in an agent system. Currently, we are finishing the implementation of a first combination of code mess-up algorithms [Röh97], and we are starting to develop a formal model of the mess-up effects to be able to compute their protection strength. To prevent blackbox testing attacks, we are currently working on an extension of the blackbox mechanism, which will also allow agents to authenticate their hosts.

3.14 Bibliography

[FGS96] Farmer, William; Guttmann, Joshua; Swarup, Vipin: Security for Mobile Agents: Authentication and State Appraisal, in: Proceedings of the European Symposium on Research in Computer Security (ESORICS), pp. 118-130, Springer LNCS 1146, 1996

[GM96] General Magic: The Telescript Reference Manual. 1996. http://www.genmagic.com/Telescript/Documentation/TRM/

[Hoh97] Hohl, Fritz: An approach to solve the problem of malicious hosts. Universität Stuttgart, Fakultät Informatik, Fakultätsbericht Nr. 1997/03, 1997. http://www.informatik.uni-stuttgart.de/cgi-bin/ncstrl_rep_view.pl?/inf/ftp/pub/library/ncstrl.ustuttgart_fi/TR-1997-03/TR-1997-03.bib

[Vig98] Vigna, Giovanni (Ed.): Mobile Agents and Security, Springer-Verlag, to appear 1998

[Röh97] Röhrle, Klaus: Konzeption, Implementierung und Analyse von Verwürfelungsmechanismen für Quellcode, Diploma Thesis Nr. 1541, Faculty of Informatics, University of Stuttgart, Germany, 1997

[Mob96] Mobilis: Exploring Telescript - mobilis Reader Interview: General Magic’s Jim White. Mobilis March 1996. http://www. volksware.com/mobilis/march.96/interv1.htm

[Mole] Mole project page. http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html

[Pal94] Palmer, E: An Introduction to Citadel - a secure crypto coprocessor for workstations, in: Proceedings of the IFIP SEC’94 Conference, 1994

[RJ96] Rasmusson, Lars; Jansson, Sverker: Simulated Social Control for Secure Internet Commerce, in: New Security Paradigms ‘96, ACM Press, 1996

[ST97a] Sander, Tomas: Security! or “How to Avoid to Breath Life in Frankensteins Monster”. Slides of a talk at the ICSI Inhouse Workshop on Auto Mobile Code, “Technology and Applications of Auto Mobile Code (AMC)”, September 1997. http://www.icsi.berkeley.edu/~tschudin/amc/workshop97/security.html

[ST98a] Sander,Tomas; Tschudin,Christian: Protecting Mobile Agents Against Malicious Hosts, in: Vigna, Giovanni (Ed.): Mobile Agents and Security, Springer-Verlag, 1998. http://www.icsi. berkeley.edu/~sander/publications/MA-protect.ps

[ST97b] Sander,Tomas; Tschudin,Christian: Towards Mobile Cryptography. Technical Report 97-049, International Computer Science Institute, Berkeley. 1997. http://www.icsi.berkeley.edu/~sander/publications/tr-97-049.ps

[ST98b] Sander,Tomas; Tschudin,Christian: On Sofware Protection via Function Hiding. Submitted to the 2nd International Workshop on Information Hiding, Dec 1998. http://www.icsi.berkeley.edu/~sander/publications/hiding.ps

[Vig97] Vigna, Giovanni: Protecting Mobile Agents through Tracing, in: Proceedings of the Third ECOOP Workshop on Operating System support for Mobile Object Systems, 1997. To appear.

4 WP 2.3: Recoverable Agents

An important prerequisite for the use of mobile agents in industrial environments is to provide reliable and fault tolerant execution of the agents. This chapter describes two different approaches to provide the required reliability. In the first approach, fault-tolerant execution of agents on unreliable systems is provided. Section 4.1 summarizes two published papers [RoSt98][StRoMa98] on this topic. In the second approach, the reliability of agents is provided by using the TUXEDO platform on Tandem Himalaya to build the agent system.

4.1 A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents

4.1.1 Introduction

Over the last few years, the concept of mobile agents has drawn a lot of attention in both academia and industry. Today many prototypes of mobile agent systems exist, most of them based on the Java programming language. Moreover, various efforts to standardize mobile agent technology are already underway (e.g., OMG MASIF, CSELT FIPA). However, despite of all these activities, only few “real” applications based on mobile agents exist today. One reason for that might be that current mobile agent platforms are in a rather early stage. Application-critical functions, such as security mechanisms, are often incomplete or missing at all. Moreover, only little work has been done so far in studying the problem of integrating agent technology with legacy systems, such as TP-Monitors and transactional resource managers. In this paper, we will show how agent technology can be integrated with transactional technology to improve fault-tolerance.

Mobile agents are autonomous objects that are able to migrate from node to node in a computer network. When an agent decides to migrate to another node, the agent’s code, data and execution state[1] is captured and transferred to the next node, where it is initiated after arrival. Agent execution proceeds in *stages* [Sch97], where the operations of a stage are performed at a single node. Whenever an agent moves to a new node, this ends the current stage and begins a new one. The assignment of stages to nodes can be defined by means of a user-defined itinerary [LO97][GM] before the agent is launched, or on the fly by the agent logic taking into account the current system state [PS97][SBH96].

The use of mobile agents has been proposed for many application areas, including electronic commerce, systems management, or active messaging. In electronic commerce scenarios, for instance, agents autonomously go shopping on a user’s behalf, do the reservations needed for a business trip, or monitor the stock market and trigger user-defined operations when certain conditions occur. Obviously, many of these applications require an agent to be executed ***exactly once***. For example, assume a user that launches a mobile agent to make a flight and hotel reservation for a forthcoming business trip. The agent is expected to make both reservations if possible, and in any case return a status message back to the user. Of course, the user will only del-

1. Actually we distinguish between *strong* and *weak migration* [GV97]. While weak migration only transfers the code and data, strong migration also transfers the agent’s execution state.

egate this job to an agent if it is guaranteed that the agent does it “exactly once” and cannot be caught by a network partitioning or node failure. In other words, independent of node and communication failures it must be ensured that the agent is never lost and hence will get its job done eventually. Moreover, failures may not cause the agent to perform operations more than once (e.g., to reserve and pay two seats instead of one).

The exactly once property has already been defined for RPC systems [Spe82], where it defines the failure semantics of a single remote procedure. In the context of mobile agents, a sequence of agent stages are to be considered rather than a single procedure. An agent execution is defined to be “exactly once” if the entire sequence of its stages is eventually performed, and all operations of each stage are executed exactly once.

In this paper, we will first describe a simple protocol based on transactional message queues (e.g. IBM MQSeries, see [BE97][Bla95]). This protocol already provides the “exactly-once” semantics as defined above. However, for many applications it is not sufficient to get the job done “eventually” but as fast as possible or even up to a certain deadline. In our reservation example above, the agent’s status message should arrive at least before the date the business trip is scheduled. The problem with our simple protocol is that an agent may be blocked due to a node crash or network partitioning even if there are other nodes, where it could continue processing. Therefore, we propose an extension of this simple protocol to reduce the probability of agents to be blocked. The extended protocol allows a number of *observer* nodes to be assigned to each stage. The observers monitor the stage node currently executing the agent and take over agent execution when this node becomes unavailable. A voting procedure integrated in commit processing ensures the “exactly-once” semantics. The protocol is currently implemented in Mole [Mole][BauEA98], a mobile agent system developed at Stuttgart University.

The remainder of the paper is structured as follows. In the next section, we will describe our agent execution model. Section 4.1.3 presents the simple protocol and discusses the problems associated with it. Section 4.1.4 introduces an enhanced model for agent processing and gives an overview of the extended algorithm, which is subdivided in a voting protocol and a so-called selection protocol. These two protocols are described in detail in Section 4.1.5 and Section 4.1.6 . Section 4.1.7 discusses the gain in fault-tolerance obtained by the protocol and gives a short estimation on the costs introduced. Section 4.1.8 presents an approach to reduce the overhead introduced by the protocol. Related work is discussed in Section 4.1.9 , before the paper concludes with a brief summary.

4.1.2 Agent Execution Model

In our ***agent execution model***, tasks are assigned to agents, which perform them autonomously. To execute its task a mobile agent may exploit the services provided by the various nodes of a computer network. According to the mobile agents paradigm [GV97] an agent moves to a node before accessing the node’s services, i.e., agents only interact with local services. Once launched an agent moves from node to node according to its *itinerary*, which may be determined before the agent is initiated or dynamically while agent execution is in progress. A more detailed description of the concept of an itinerary can be found in [StRoMa98].

Agent execution proceeds in ***steps***, where a new step is initiated whenever an agent migrates to the next node. A step of an agent at a node is defined to be the set of operations performed by

the agent while it visits this node. Consequently, all operations of a given step are performed at the same node and access local resources only. In our model, we assume that resources are encapsulated in resource managers, which - depending on the actual system environment - may be represented by stationary agents, servers or (recoverable) objects. Each step may change the agent's state as well as the state of the local resources. For example, assume an agent buying a ticket from a ticket server. After this step, the agent's state would reflect the ticket information as well as the modified electronic wallet data, and the ticket server's database would have been updated accordingly. Note that this step has to be performed in an atomic manner.

Let $L(I)$ be the number of nodes in the agent's itinerary $I=[N_1,N_2,...,N_{L(I)}]$ and S_i be the step to be performed on node N_i ($1 \le i \le L(I)$). Then the execution of an agent is defined to be exactly-once if

- the agent executes step S_i before step S_{i+1}, $1 \le i < L(I)$, and
- each step S_i $1 \le i \le L(I)$ is executed exactly once, independent of communication and node failures.

As we will see below, the exactly-once semantics of steps is implemented by means of ACID transactions in conjunction with a mechanism that guarantees a step transaction to be performed exactly once.

For the protocols described in this paper, we will assume the following ***system model***. Nodes are interconnected by means of a communication network, and each node has volatile as well as stable storage [Lam81]. Moreover, nodes are assumed to suffer from crash failures [Jal94] only. Communication failures may cause the network to be partitioned. The communication network provides for reliable channels provided the sender and receiver reside in the same partition, i.e., messages are not lost, garbled or duplicated and are delivered in order.

4.1.3 A Simple Solution

The exactly-once property of mobile agents as defined above can be achieved in a simple way by using transactional message queues (e.g., see [GR94]). Message queues provide for asynchronous communication between processes residing on the same or different nodes, where the sender of a message *Puts* this message on a queue and its receiver *Gets* it from that queue. Transactional message queues provide for persistent messages and ensure the exactly-once delivery, i.e. once a queue manager has accepted a message, it will be delivered once, independent of node and communication failures. Moreover, the *Put* and *Get* operations can be performed within ACID transactions [HR93]. A message is only placed on or removed from a queue if the transaction including the corresponding *Put* respectively *Get* operation is committed. Transactional message queues are supported by a wide range of middleware products (e.g., see IBM MQSeries, TUXEDO[GR94], Encina[GR94]).

Figure 4.1 depicts how transactional message queues can be used to implement agents with exactly-once property. Assume that an agent moves from node to node along route $N_1 \rightarrow N_2 \rightarrow ... \rightarrow N_{k-1} \rightarrow N_k$. As an agent may visit the same node several times, N_i and N_j ($1 \le i,j \le k$) may denote the same or different nodes. Assume further that an agent is stored in a message queue when it is accepted by the agent system for execution. Once the agent has been stored in the initial queue

(Q_1 in our example), the owner of the agent can be informed that this agent - provided that nodes, queues and the network recover - will be performed exactly once eventually.

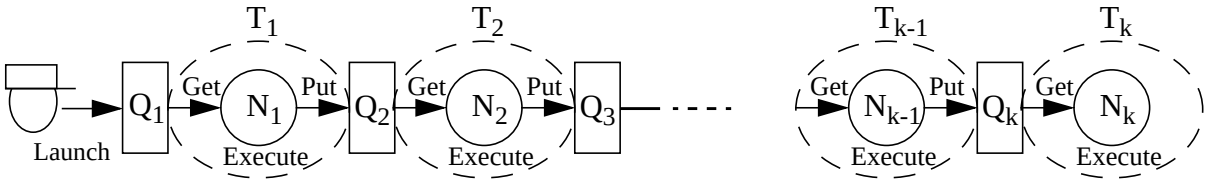


Figure 4.1 Simple implementation of exactly-once agents using message

Except N_k each other node is performing the following sequence of operations: *Begin_Transaction*; *Get*(Agent); *Execute*(Agent); *Put*(Agent); *Commit*. *Get* removes an agent from the node's input queue, *Execute* performs the received agent locally, and *Put* places it directly into the input queue of the node to be visited next. All three operations are performed within a transaction and hence build an atomic unit of work. So, if for instance transaction T_i aborts due to a node or transaction failure, recovery undoes all of the agent's effects at N_i and restores the agent in its original state in Q_i. Any effects in Q_{i+1} are undone also. After recovery is finished, N_i continues normal processing and will execute this agent eventually and then hand it over to its successor. Of course, the last node in the agent's itinerary, N_k, does not have to perform *Put*, it simply destroys the agent when the execution is finished.

The problem with this simple solution stems from the autonomy of agents. Due to this property there is no "natural" instance that monitors the progress of an agent. If a node crashes after the agent has been placed in its (local) input queue and before it is moved to the next queue, the agent is "caught" as long as the node is down even if there are other stage nodes which could execute the agent. A partitioning of the underlying network may have similar effects. Note that this is different in client/server systems, where a client calling the operations of a server monitors the availability of this server. When it detects a server failure, the client can continue processing by using alternative servers offering the same or similar services.

With the above protocol, there is no system entity that will notice that an agent is "caught". Of course, the end user might notice when the agent misses a deadline. This is a serious drawback since agent processing is blocked even if alternative nodes providing the needed services are available. Even if those nodes do not exist, some sort of exception handling should be performed, e.g., informing the user that the agent will most probably miss the deadline. In the next section we will extend the simple protocol described above to reduce the probability of agents to be blocked due to failures.

4.1.4 Protocol Overview

The execution of an agent proceeds in a sequence of ***stages***. The operations associated with a stage are entirely performed at a single node, and an agent enters a new stage whenever it moves to the next node. For each stage there exists a non-empty set of nodes which alternatively can perform that stage. Each stage initially includes a ***worker*** node, which is responsible for executing the agent in this stage. The other nodes are ***observers*** monitoring the availability of the stage's worker. When the worker fails, this will be detected by the observers, which then will elect a new worker from the set of available stage nodes. Each stage node is associated with a

priority, which defines a total ordering between the nodes belonging to the same stage. Node priorities are required for the voting and selection process. The node with the highest priority becomes the initial worker of a stage. Figure 4.2 shows a 3-stage execution of an agent. For example, stage S_1 is associated with one worker, and 4 observers. In S_2, the node with the highest priority (1) failed and the node with priority 2 was elected to be the new worker.

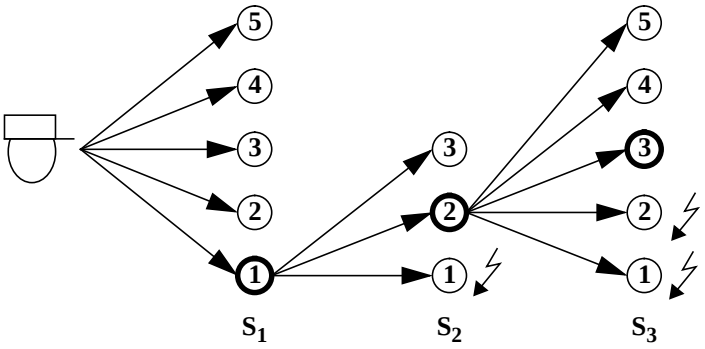


Figure 4.2 Execution of an agent in 3 stages

What are the functional capabilities expected from observers? Ideally, an observer provides the same set of services an agent expects to find at the initial worker (e.g., a flight reservation service). However, an observer that offers no more than an environment for running agents is also acceptable. At such a node an agent can perform the exception handling mentioned above. For example, it can use the infrastructure services to find alternative servers, it can change its travel plans, or it can just move back to the user's machine to report the problems and receive new directions.

To allow an observer to take over agent execution, it obviously needs a copy of the agent. Therefore, in our scheme, a worker sends the agent not only to the (initial) worker but to all nodes of the next stage when it has finished processing. However, only the worker initiates agent processing, while the observers just do the monitoring for this stage. As in our simple protocol above, we use transactional message queues to move agents from one stage to another. In contrast to the simple protocol, the Put-operation of the message queue has to ensure that the message has already arrived at the destination node at transaction commit. Stage processing has the following structure (see Figure 4.3): *Begin_Transaction*; *Get*(Agent); *Execute*(Agent); *Put*(Agent) *to* (AllNodesOfNextStage); *Commit*.[2]

The monitoring protocol (see Section 4.1.6) ensures that an observer eventually recognizes when a worker becomes unavailable. In such a case, the observers of that stage select a new worker, which initiates a *new* stage processing transaction comprising the sequence of operations described above. Now, there is an obvious problem with this approach. Since the observers in general cannot decide whether an unavailable worker has crashed or is still active in a different partition of the network, it may happen that two or more nodes of the same stage execute the

2. Note that the *Get* operations of the observer nodes are not part of the transaction. If they would be included in the transaction, this would require all nodes of a stage to be available to execute an agent at that stage. Clearly, this increases the probability that an agent becomes "caught" rather than decreasing it. It is important to notice that having several *Puts* instead of one in the transaction does not increase the "caught" probability since the observers for the next stage can be determined on the fly from the set of available nodes.

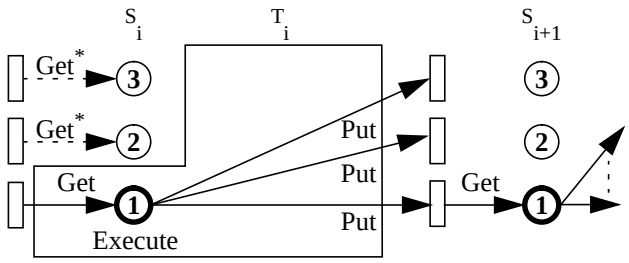


Figure 4.3 The transactional processing of an agent in a stage

agent at the same time. However, the exactly once property of agents requires that exactly one stage transaction is committed per stage. In order to achieve this, we integrate a voting protocol into the two-phase commit (2PC) processing [GR94] of stage transactions: a transaction can only commit if a majority of stage nodes agree. It is also the responsibility of this voting protocol to make sure that all observers of a stage remove all stage information when a worker's stage transaction commits (see the *Get** operations in Figure 4.3).

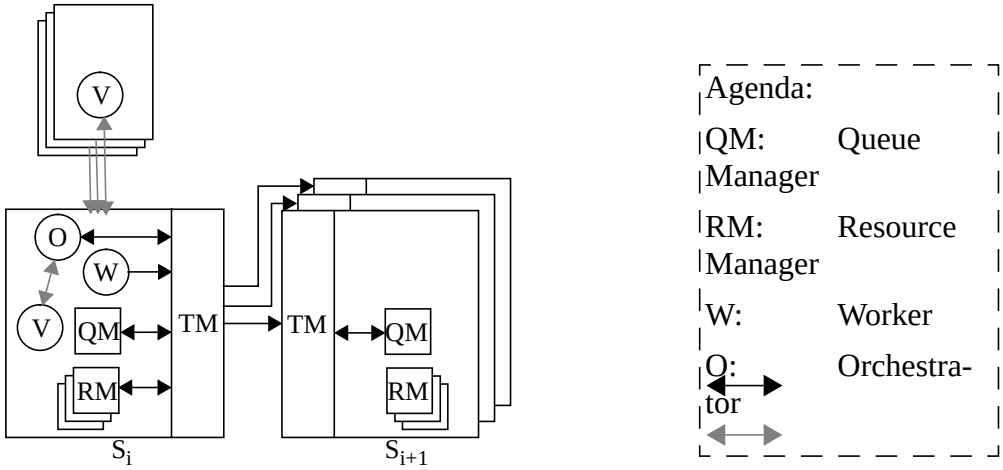


Figure 4.4 Components and interactions relevant to process a stage

In Section 4.1.5 , we will present our voting protocol and show how it can be integrated in standard 2PC processing. We will assume an architecture similar to the X/Open Distributed Transaction Processing [X/O91] architecture, which consists of transaction managers running the 2PC protocol and resource managers maintaining the recoverable data. Figure 4.4 depicts the components and interactions relevant for the processing of stage S_i. When the worker of S_i calls *Commit*, the local TM initiates 2PC processing, which involves the worker itself and all nodes of S_{i+1}. During the commit procedure, each involved TM interacts with those local resource managers that were involved in stage processing. For example, at the worker node this is the queue manager associated with the worker's input queue and the other local resource managers (e.g. a DBMS) that have been involved in agent execution. In addition, the worker's TM interacts with another type of resource manager called ***orchestrator***. The orchestrator, which communicates with the so-called ***voters*** belonging to its stage, is responsible for orchestrating the voting procedure. Each stage node runs a voter, which determines and communicates the node's

vote. The orchestrator and the voters of a stage communicate according to the voting protocol presented in the next section.

It is important to notice that the proposed architecture nicely separates voting and 2PC processing. From a TM's point of view, the orchestrator is just another resource manager, which provides the same interface as all other resource managers (e.g., an XA interface [X/O91][BE97]). Consequently, the voting procedure can be easily integrated in existing middleware systems, such as CORBA [OMG96] or X/Open compliant systems, just by implementing a new resource manager, or a new recoverable server to use CORBA terminology.

Besides the voting procedure, a ***selection protocol*** is needed, which allows the observers of a stage to select a new worker when they recognize that the old one failed. Since the voting during 2PC processing already ensures that only one stage transaction commits, the exactly once property is not jeopardized even if more than one new worker is selected. Actually, each observer that recognizes a worker failure could select itself without talking to the other observers. Consequently, the problem of selecting a new worker differs from the well-known election problem as defined in the literature (e.g., see [GM82]). For that reason we are using the term "selection" rather than "election" throughout this paper.

The selection protocol proposed in Section 4.1.6 is a "light-weight" protocol, which usually selects one new worker, but also can end up with multiple workers in rare situations. Each worker and observer node runs monitor processes that do the monitoring and the selection of new workers if needed.

4.1.5 Voting Protocol

In this section, we will focus on the voting protocol and its integration into 2PC processing. Instead of describing the well-known 2PC procedure, we will confine ourselves on presenting the interactions between the transaction manager (TM) and the local voting orchestrator (see Figure 4.5).

The voting protocol used here is based on the fault tolerant majority quorum algorithm [Thom79][Giffor79] and is extended by an algorithm similar to the one described in [Maek85] to resolve concurrent invocations of the algorithm using priorities. In terms of message complexity more efficient quorum based fault-tolerant algorithms for mutual exclusion have been proposed (e.g. [AgrAbb91], [ChaCha97]). Although these algorithms have the same time complexity in the error-free case (O(1)), they have a higher time complexity in the presence of failures. In addition, the difference in number of messages for small stages (5 to 7 nodes) is marginal in the error free case. Therefore, we chose to use the simple algorithm.

As already stated in the previous section, from the TM's point of view the orchestrator looks like an ordinary resource manager. We assume that resource managers implement an XA-like interface with the following operations: *rm_prepare*, *rm_commit* and *rm_rollback*. The first operation, called in the first phase of 2PC, returns either *rm_yes* or *rm_no*, depending on whether or not the resource manager is able to prepare for commitment. In the second phase, the TM issues either *rm_commit* or *rm_rollback* depending on the transaction's outcome. Upon such a call a resource manager terminates the transaction accordingly and returns *rm_ack* to the TM.

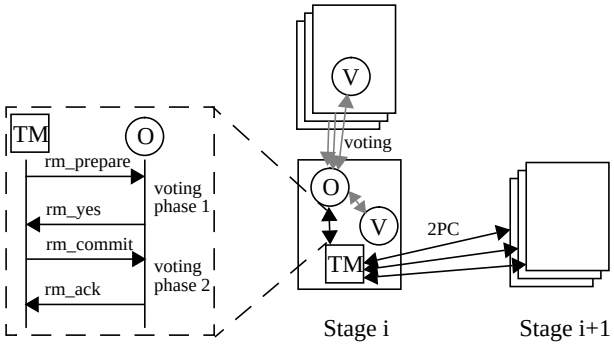


Figure 4.5 Integration of voting into 2 PC

The interactions between the 2PC of the stage transaction and the voting protocol are shown in Figure 4.5.

The voting protocol is run between the orchestrator and the voters of a stage. While the orchestrator is located at a worker node only, there exists a voter at each stage node. When 2PC processing is started at the orchestrator (i.e., when *rm_prepare* is called), it issues vote requests to the voters of its stage and then collects the returned votes. Only if it receives a majority of yes votes, the orchestrator returns a *rm_yes* to its local TM, and a *rm_no* otherwise. In other words, only if a majority of voters vote yes, the transaction can be committed. That is why only one transaction can commit per stage even if there is more than one worker.

We distinguish between two types of stable states, namely ***transaction states*** and ***stage states***. Both are stored on stable storage and thus are supposed to survive node failures. Transaction states are maintained by orchestrators, while voters maintain stage states. A transaction's state can be "Unknown", "Ready" or "Committed", while a stage's state may be "Unknown" or "Active". For both types of states "Unknown" means that no state information is stored on stable storage for the corresponding transaction or stage. The state information of an "Active" stage is stored in a so-called ***stage record*** on stable storage. It contains the following information:

- An identifier of the stage, which consists of an (*AgentId*, *HopCount*) pair. *AgentId* is a globally unique agent identifier and *HopCount* is incremented whenever the agent is moved to the next stage.
- A list of nodes participating in the stage. For each node the node's identifier and priority is included.

When an agent moves to the next stage, not only the agent itself but also the stage record of the next stage is *Put* into the input queues of the nodes associated with the next stage. Each stage node reads the stage record without actually removing it. Once the stage record has been *Put* into the message queue (on stable storage), the stage becomes "Active" at the corresponding node. Since all *Put* operations are performed in a single transaction (see Figure 4.3), either all stage nodes are "Active", or none of them. Initially, the stage node with the highest priority becomes the worker, while all other nodes take over the observer role.

Voters and orchestrators are identified by globally unique node identifiers (ids), i.e., a voter and orchestrator residing on the same node have the same id. In analogy to 2PC processing, our voting protocol proceeds in two phases which are described in the following paragraphs. Figure 4.6 shows two possible scenarios in a stage with two stage nodes to illustrate the voting algorithm.

In Figure 4.6a only one orchestrator initiates the voting procedure while Figure 4.6b shows a scenario where both orchestrators initiate the voting procedure concurrently.

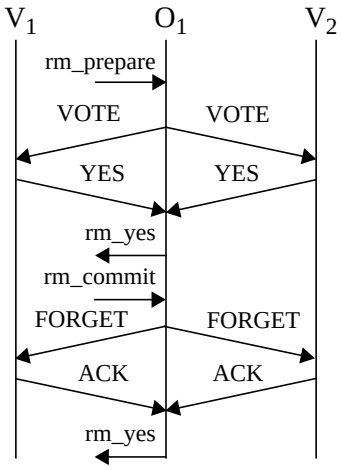


Figure 4.6a One orchestrator initiating voting

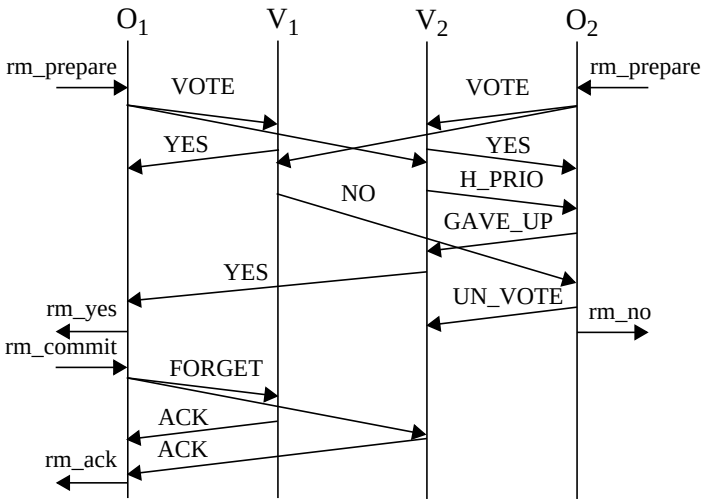


Figure 4.6b Two orchestrators initiating voting concurrently

Normal Processing: Phase 1

Phase 1 of the voting protocol is initiated when an orchestrator receives a *rm_prepare* call from its local TM. First, the orchestrator sends a VOTE request to each voter of its stage. This request includes several globally unique identifiers: the id of the stage currently processed, the orchestrator's id, and the id of the transaction the orchestrator is currently involved in[3]. Then, the orchestrator waits for the answers, periodically resending the VOTE request to all stage nodes that have not yet answered.

To record its votes already given to orchestrators, a voter maintains a list called *OrchSet* on stable storage. Whenever the voter returns a YES vote, the identifier of the receiving orchestrator is recorded in *OrchSet*. Normally, *OrchSet* ends up with one node identifier. In the presence of failures, however, there might be several orchestrators competing for a node's vote.

A voter receiving a VOTE(*StageId, TId, OrchId*) request for stage *StageId* determines its reply based on its *OrchSet*. If *OrchSet* is empty, the voter has not voted YES before. In this case, *OrchId* is added to *OrchSet* and a YES(*StageId, TId, VoterId*) reply is sent back to the orchestrator, where *VoterId* identifies the voter (replies of the voters in Figure 4.6a, the first replies of the voters in Figure 4.6b).

If *OrchSet* is not empty instead, there are obviously several orchestrators competing for the vote. To make sure that one of the them will eventually receive a majority of votes, our voting protocol prefers the orchestrator with the highest priority. Assume that *N* is the node with the highest priority in *OrchSet*. If *OrchSet* is not empty and *OrchId* has a lower priority than *N*, then the voter

3. The transaction identifier is received in the *rm_prepare* call and is used here to match VOTE requests with the corresponding votes. Due to node failures it may happen that the same orchestrator starts several rounds of voting.

has already voted YES for a node with a higher priority. In this case, the voter replies with NO(*StageId, TId, VoterId*), i.e., *OrchId* loses the competition. In our scenario in Figure 4.6b, V_1 already gave its vote to orchestrator O_1 which has a higher priority than orchestrator O_2. That is why V_1 sends a NO to O_2.

If *OrchSet* is not empty and *OrchId* has a higher priority than *N*, then the voter has already voted but only for orchestrators with a lower priority. If *N* is not the voter's node, the voter immediately sends back a COND_YES(*StageId, TId, OrchSet, VoterId*) and then adds *OrchId* to its *OrchSet*. The semantics of this vote is that *VoterId* votes YES, provided that all nodes in *OrchSet* also vote YES.

If *N* equals the voter's node, there exists a local orchestrator, which has already initiated a competing voting procedure. Since *OrchId* has a higher priority than the local orchestrator, the latter one is supposed to give up. This, however, is only possible (and desirable) before the stage transaction at the orchestrator has entered the "Ready" state (i.e., before the orchestrator has got a majority of votes). To check the transaction's state, the voter sends a HIGHER_PRIO request to the local orchestrator, which returns either GAVE_UP to indicate its stage transaction has been aborted, or ALREADY_DONE if the transaction state is already "Committed" or "Ready". If ALREADY_DONE is returned, the voter sends a NO(*StageId, TId, VoterId*) message to *OrchId*, and a COND_YES(*StageId, TId, OrchSet-{N}, VoterId*) message (or YES(*StageId, TId, VoterId*) if *OrchSet-{N}* is empty) otherwise. In Figure 4.6b, V_2 receives a vote request from O_1 after already having given a YES vote to O_2 which has a smaller priority than O_1. Therefore, V_2 sends a HIGHER_PRIO request to O_2 which, still being in "Unknown" state, replies with GAVE_UP, enabling V_2 to reply a YES vote (COND_YES with empty OrchSet) to O_1.

To record the received votes matching the current *TId*, the orchestrator maintains three sets in volatile storage: *YesVotes*, *NoVotes* and *CondYesVotes*. When it receives a YES or NO vote, it includes the voter's id in *YesVotes* or *NoVotes*, respectively. If it receives a COND_YES, it adds the (*VoterId, OrchSet*) pair included in this message to *CondYesVotes*. Note that this conditional YES becomes a "real" YES after all nodes in *OrchSet* voted YES. In other words, if *OrchSet-YesVotes* equals the empty set, *VoterId* can be added to *YesVotes* and (*VoterId, OrchSet*) can removed from *CondYesVotes*. Obviously, this check has to be performed when the (*VoterId, OrchSet*) pair is added to *CondYesVotes* and whenever *YesVotes* is changed.

Once *YesVotes* contains a majority of votes, the orchestrator moves into the "Ready" state and then returns *rm_yes* to its local TM (see orchestrator O_1 in our scenarios). Then it waits for the TM's commit or abort decision. Note that the *rm_yes* response is only a prerequisite for commitment rather than a commit decision. If a majority becomes impossible (i.e., at least half of the voters voted NO), the orchestrator returns a *rm_no* to its local TM, sends an UN_VOTE(*StageId, TId, OrchId*) message to all voters recorded in its *YesVotes* and *CondYesVotes* set, and then forgets the transaction (see orchestrator O_2 in Figure 4.6b). Note that the *rm_no* response forces the TM to abort the stage transaction. The orchestrator's node then changes from the worker to the observer role (see Section 4.1.6).

When the orchestrator receives a HIGHER_PRIO message, it replies ALREADY_DONE if its stage transaction is already "Ready" or "Committed". If the transaction is still in the "Unknown" state, it sends back GAVE_UP to the local voter and returns *rm_no* to the local TM (see O_2 in Figure 4.6a). Furthermore, it sends UN_VOTE messages to all voters recorded in its *Yes-*

Votes and *CondYesVotes* set before it forgets the transaction. As above, orchestrator's node then changes from the worker to the observer role.

An orchestrator receives a GIVE_UP request from the local voter if another stage node already committed its stage transaction (see below). Clearly, this message can only arrive while the receiving orchestrator resides in the "Unknown" transaction state. When GIVE_UP arrives, it immediately forgets the transaction, and returns *rm_no* to the local TM.

Normal Operation: Phase 2

If the TM commits the transaction, it issues *rm_commit* for each local participating resource manager. When *rm_commit* is called, the orchestrator atomically enters the "Committed" state and returns an *rm_ack* to the local TM. Subsequently, it sends a FORGET(*StageId*, *OrchId*) message to all voters of its stage and then waits for the acknowledgements to arrive. It periodically resends FORGET until it received an ACK from each voter. When the ACKs are complete, it moves to the "Unknown" transaction state before it forgets the transaction.

A voter receiving FORGET atomically goes into the "Unknown" stage state, i.e. the stage's stage record (together with the agent) is removed from the voter's transactional input queue in an atomic fashion. Subsequently, the voter removes the stage's *OrchSet* from stable storage and sends back an ACK(*StageId*) message to the sender of FORGET. If there happens to be a local orchestrator different from *OrchId*, the voter sends GIVE_UP to this orchestrator, causing the locally initiated stage transaction to be aborted.

If the orchestrator receives *rm_abort* instead of *rm_commit* from its TM, it enters the transaction state "Unknown" and then sends UN_VOTE requests to all voters recorded in its *YesVotes* or *CondYesVotes* set. Then the orchestrator's node restarts the transaction. Voters receiving an UN_VOTE remove *OrchId* from their *OrchSet*, i.e., they withdraw their votes previously given to *OrchId*. Note that this "unvote" mechanism is needed to allow a lower priority node to achieve a majority after some higher priority node gave up. Remember that a voter only votes YES (or COND_YES) if it has not already voted YES (or COND_YES) for some other node with a higher priority

Failure Recovery

Once a voter has returned a vote to an orchestrator, it can expect either a FORGET or UN-VOTE response. When the voter times out while waiting on the response, it sends an INQUIRY message to the corresponding orchestrator. INQUIRY messages are sent periodically until FORGET (or GAVE_UP) is received, or each orchestrator recorded in the voter's *OrchSet* returned an UN_VOTE response.

An orchestrator's response on an incoming INQUIRY(*StageId*, *VoterId*) request depends on its current transaction state. If the transaction state is "Ready", the orchestrator adds *VoterId* to its *YesVotes* set if it is not already included. This ensures that the identified voter will be notified accordingly as soon as the TM issues *rm_commit* or *rm_abort*. If the orchestrator is in the "Committed" state when receiving an inquiry, it responds with a FORGET message. If the orchestrator resides in the "Unknown" state, two cases must be distinguished: If there is no active

transaction belonging to the stage identified by *StageId*, the orchestrator returns an UN_VOTE message. If there is an active transaction instead (i.e., voting is still in progress for the stage) the INQUIRY can be ignored. Let us briefly argue why. If *VoterId* is already in *YesVotes* or *CondYesVotes* or will be included at a later point in time, the identified voter will be informed during phase 2. Even if this is not the case, a future INQUIRY will eventually find no locally active transaction, causing an UN_VOTE to be returned to the voter.

When a node recovers from a failure, it reads the transaction and stage states recorded in stable storage. Orchestrator recovery only takes place if the transaction state is "Committed" or "Ready". If the transaction is "Ready" after restart, the orchestrator waits until it is informed by the local TM about the transaction's outcome, and then proceeds as described above. If the transaction is already "Committed" instead, the orchestrator sends FORGET to all voters of the stage and collects the ACKs. After having received all ACKs, it can enter the "Unknown" transaction state.

A voter only performs recovery if its stage state is "Active". In this case, the voter periodically sends INQUIRY request to all orchestrators recorded in its *OrchSet.* It continues to send inquiries until it receives FORGET from some orchestrator, or it got an UN_VOTE from each orchestrator in *OrchSet*. It acts upon the received responses as described above.

Correctness Arguments

In the following, we will give some informal correctness arguments for the voting protocol described above. We will assume that the selection protocol ensures that there eventually exists a non-empty set of orchestrators (or workers). The objective of the voting protocol is to guarantee that exactly one of these orchestrators will commit its state transaction.

Let us first show that - given a non-empty set of orchestrators - exactly one of them will eventually enter the "Ready" transaction state. If there is only one orchestrator, it will get YES votes from all available voters. As soon as a majority of voters is available, it can enter the "Ready" state.

Now assume that there are several competing orchestrators, and O_1 is the one with the highest priority. When another orchestrator, say O_2, receives O_1's vote request, it has either already entered the "Ready" or "Committed" state, or it gives up. In the first case, O_2 has got a majority of votes, which allows no other orchestrator to move into the "Ready" state. In the latter case, O_2 sends UN_VOTE to its voters, allowing its local voter to send a YES vote back to O_1. All other voters either return a YES or COND_YES (O_2) message back to O_1, depending on the sequence O_1's and O_2's vote requests arrived. Since O_2 returned a YES vote, the COND_YES(O_2) votes can be interpreted as yes votes. Consequently, O_1 will eventually receive a majority of votes and thus can enter the "Ready" state.

If O_1 aborts its transaction, the "unvote" mechanism ensures that all voters will eventually withdraw their votes given to O_1 (or forget the stage). Therefore, also a lower priority orchestrator will get the chance to collect a majority of votes.

As shown above, if there are several orchestrators, exactly one of them will eventually enter the "Ready" state. This orchestrator's transaction will either commit or abort. In the case of commitment, all stage nodes forget the stage, and thus no other transaction of this stage will be able

to commit any more. In the case of abort, the transaction becomes “Unknown” and its orchestrator starts a new transaction. In the latter case, as shown above, this or another orchestrator will eventually become “Ready”. Consequently, exactly one orchestrator will eventually perform commitment.

4.1.6 Monitoring and Selection Protocol

In the previous section, we already pointed out that in addition to the agent also the stage record of the stage to be performed next, say *S*, is *Put* into the input queues of the nodes associated with *S*. Remember that all these *Put* operations are performed within the transaction of the previous stage and thus are “all or nothing”. Each stage node reads the stage record without removing it from its input queue and decides its initial role depending on the priorities recorded in the stage record. The node with the highest priority becomes the worker node, which then performs the sequence of operations already outlined in Section 4.1.4 : *Begin_Transaction*; *Get*(Agent); *Execute*(Agent); *Put*(Agent, StageRecord) *to* (AllNodesOfNext-Stage); *Commit*. The other stage nodes are observers, which monitor the worker.

A worker, say *W*, periodically sends I_AM_ALIVE messages to the observers of its stage. If it receives an I_AM_ALIVE or I_AM_SELECTED (see below) message from another node, then there obviously exists a competing worker, say *W*’. If *W*’ has a higher priority than *W*, *W* sends a HIGHER_PRIO request to the local orchestrator. If the response is GAVE_UP (see Section 4.1.6), *W* becomes an observer monitoring *W*’.

When an observer times out while waiting on the worker’s I_AM_ALIVE messages, it assumes that the worker is not available any more and initiates the procedure for selecting a new worker. The selection protocol described below adopts the basic principles of the fault-tolerant bully algorithm [GM82], which allows elections taking place even in presence of network partitioning. As already mentioned above, this protocol may end up with several workers in case of network partitioning. The voting protocol described in 4.1.5 ensures that only one worker commits its transaction.

A node initiating the selection procedure sends ARE_YOU_THERE messages to all stage nodes with a higher priority. Available nodes (observers as well as workers) reply to this message with an I_AM_THERE message. If no reply arrives within a reasonable time, the initiator is selected to be the new worker (Figure 4.7a). The newly selected worker sends an I_AM_SELECTED message to all other stage nodes, and starts a new stage transaction comprising the sequence of operations sketched above. If the initiator receives a reply instead, it waits for the I_AM_SELECTED (or I_AM_ALIVE) of the new worker to arrive (Figure 4.7b). When this message arrives, it starts monitoring the new worker.

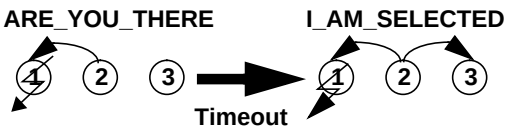


Figure 4.7a Successful selection

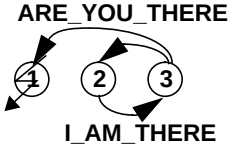


Figure 4.7b Unsuccessful selection

In the presence of network partitioning, the protocol presented so far selects a worker in each partition, if two partitions are joined, two workers remain in the resulting partition. Note that this is not a problem since our voting protocol ensures that only one worker will commit.

Starting a transaction in a partition that does not include a majority of nodes is at least questionable. With a little modification of the protocol, starting transactions in partitions without a majority of nodes can be avoided: Observers getting an I_AM_SELECTED message are supposed to reply with an ACK, and the initiator of the selection protocol becomes the new worker only if it receives a majority of ACKs. Therefore, the initiator periodically sends I_AM_SELECTED messages until it receives either a majority of ACKs or an I_AM_SELECTED from a higher priority node. In the first case, it becomes the new worker, while it continues to be an observer in the latter case.

The worst case message complexity of the chosen algorithm is $O(n^2)$ (all n stage nodes initiate the algorithm at the same time), the time complexity of the algorithm is O(1). A lot of other, more efficient election algorithms have been proposed (e.g. [AA88], [MNHT89], [Singh96], [Singh97]). Most of these algorithms do not terminate in the presence of network partitioning, but the main idea of some of these algorithms, to reduce message complexity by "eliminating" the other nodes sequentially, could be applied to our selection problem. Using this approach enhances the time complexity considerably to $O(n)$. Therefore, as long as stages are relatively small (max. 5 to 7 nodes), the chosen algorithm is considered to be convenient.

4.1.7 Blocking Probability and Message Complexity

The worker node needs to collect a majority of votes during 2PC processing to be able to commit the transaction of a stage. Therefore, a transaction can only be committed if more than half of the stage nodes (including the worker) are available. This fact can be used to give a (simple) metric for the availability A_s of a stage which is the probability that a majority of stage nodes is available so that an agent can finish a step and proceed with the next step.

Let n be the number of the nodes of a stage and p be the availability of an individual node (i.e. the probability that the node is available). Then the probability that exactly m out of these n nodes are available can be calculated using the binomial probability function $f(n, m) = \binom{n}{m} p^m (1-p)^{(n-m)}$ [HuGr71]. The availability $A_S(n, p)$ of a stage S can then be calculated by

$$A_s(n, p) = \sum_{i = \left\lceil \frac{n+1}{2} \right\rceil}^{n} \binom{n}{i} p^i (1-p)^{(n-i)} \quad .$$

The blocking probability is defined to be the probability that the agent is blocked in the stage. It is calculated by $B_s(n, p) = 1 - A_s(n, p)$. The relative blocking probability $B_r(n, p)$ is calculated by $B_r(n, p) = B_s(n, p) / B_s(1, p)$, where a relative blocking probability of $B_r(n, p)$=0.4 means for example that the probability of an agent blocking in a stage with n nodes (node availability p) is only 40% of the probability of an agent blocking on one node with availability p.

Table 4.1 and Figure 4.8 show the relative blocking probability B_r depending on the availability p of a node and the number n of stage nodes. It shows that an odd number of nodes bigger or equal to 3 reduces the relative blocking probability dramatically.

n	p		
	0.75	0.9	0.99
1	100%	100%	100%
2	175%	190%	199%
3	62%	28%	3%
4	105%	52%	6%
5	41%	9%	~0%
6	68%	16%	~0%
7	28%	3%	~0%

Table 4.1 Relative blocking probability of a stage

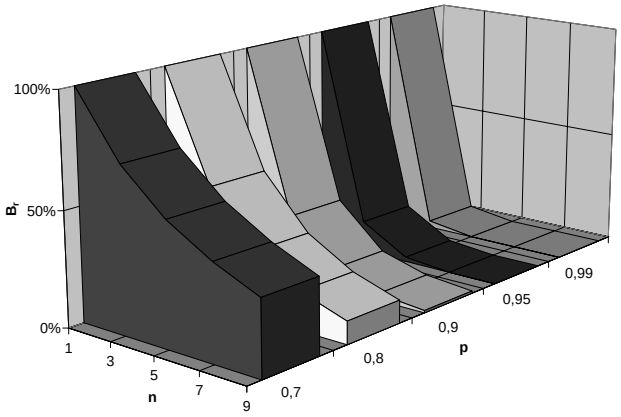


Figure 4.8 Relative blocking probability of a stagef

Obviously, the fault-tolerant protocol considerably reduces the blocking probability of an agent compared to the simple protocol of Section 4.1.3 . The price one has to pay for this is the overhead introduced by the protocol. In this section, we compare the number of inter-node messages necessary for the simple protocol with the number of intra-node messages necessary for the fault tolerant protocol. Assuming that errors are the exception, we only examine the error-free case.

The simple protocol introduced in Section 4.1.3 is the “cheapest” possibility to provide the exactly-once property for mobile agents. The amount of messages used depends heavily on the implementation of the message queues. An optimized version of a message queue only needs 4 messages, piggybacking the PREPARE to the put message: (data + PREPARE), PREPARED, COMMIT and ACK

The communications potentially taking place in the fault-tolerant protocol are shown in Figure 4.9. Using optimized message queues as described above, all in all 4*n* messages (*n*=number of nodes) are necessary for the transport of the agent to the next stage and the 2PC (solid arrows). If the worker of the current stage is also member of the next stage, the number of messages reduces to 4(*n*-1). The two phases of the voting protocol (dashed arrows) need 2(*n*-1) messages for the voting and 2(*n*-1) messages for the termination of the stage. In addition, (n-1) messages are sent periodically during stage execution for monitoring purposes.

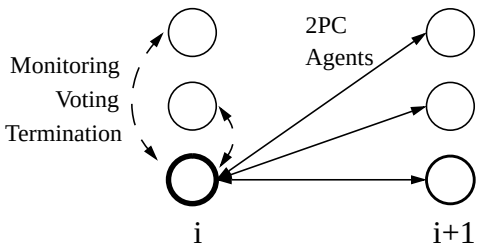


Figure 4.9 Communication patterns

Further optimizations are possible. If the timeout for monitoring is bigger than the execution time of an agent in a stage, no monitoring messages are necessary (the FORGET replaces the I_AM_ALIVE). Additionally, the FORGET message may be delayed (and piggybacked onto another message) until the next monitoring message would have to be sent. Finally, the ACK message acknowledging the FORGET can be delayed some time (it just has to be in time before

the receiver resends the FORGET). Another possibility to reduce the overhead based on the knowledge on the agents itinerary is presented in the next section.

Altogether, the number of messages for the fault-tolerant protocol without monitoring messages is between 6(n-1) if the worker is member of the next stage and all FORGETs as well as all ACKs could be piggybacked and 4n+4(n-1) if none of the optimizations is possible.

4.1.8 Optimizing the Stage Construction

The fault-tolerant protocol described above introduces some overhead. This overhead can be reduced if some information about the agent's travel plan is known in advance. The first part of this section introduces a facility called *itinerary* which allows a very flexible description of the agent's travel plan. Then the notion of the exactly-once property is extended to the itinerary concept and a classification of stage node types is introduced. Finally, an algorithm reducing the protocol overhead is presented.

The Itinerary

While performing a job, a mobile agent often has to visit several nodes to use services offered locally. In many cases, some (or all) of these nodes are either known before agent initialization or can be determined by the agent several steps in advance. However, as in real life, no strict order exists in which the nodes have to be visited. For example, an agent having to buy a CD, one pound of beef and a theatre ticket, may perform these tasks in any sequence. On the other hand, if there are several branches of a music shop, the agent needs only to visit one of these branches. To exploit the possible benefits given by a flexible travel plan (e.g. by calculating the shortest path) and to provide a powerful facility to the agent developer, an *itinerary concept* is provided. This *itinerary concept* allows a very flexible specification of an agent's travel plan as well as the dynamic adaptation and expansion of the travel plan during the execution of the agent.

The *itinerary* is composed using different types of *itinerary entries*. The simplest form of an entry is a simple pair (*node, method*) specifying a node which has to be visited and the step (defined by *method*) which has to be executed on this node (see [WoEA97]). The other possible entries, called *sequence*, *set* and *alternative* contain several other entries (recursively). A *sequence* is a list [e_1,...,e_n] of n entries ($n \geq 1$) defining that the nodes specified by entry e_i ($1 \leq i < n$) must have been visited before the nodes of entry e_{i+1} are visited. A *set* is a set of entries {e_1,...,e_n} specifying that the elements e_1,...,e_n can be handled in any order as long as each element is handled exactly once. An *alternative* (e_1,...,e_n) allows to specify that exactly one of the entries e_1,...,e_n have to be chosen.

To clarify this definition, let us consider the following scenario. Paul, planning to spend a romantic evening with his wife, instructs his personal concierge agent to order some flowers, to buy a ticket for the theatre and to reserve a table in a nice restaurant close to the theatre. The play, for which the agent has to buy tickets is currently enacted in two different theatres. To fulfil the job, the agent has to visit the node of the flower service, one of the two nodes offering the ticket service (unfortunately, there is no central ticket service for both theatres), and, depending on the chosen theatre, the node of the restaurant.

The itinerary *i* specifying the travel plan of our concierge agent is defined using the notation introduced above by

```
i = {  (BestFlowers, buyFlowers),
       (   [ (CentralTheatre, buyTicket), (KingsInn, reserveTable) ],
           [ (ModernArts, buyTicket), (BeefHouse, reserveTable) ]
       )
    }.
```

A graphical representation of the itinerary is shown in Figure 4.10a. The top level entry of the itinerary is a *set* specifying that the agent has to go and buy flowers on node "BestFlowers" (using the method buyFlowers) and to follow the specification in the *alternative*. This can be performed in any order. The *alternative* specifies the two possible ways of buying a theatre ticket and making a reservation for a table. Each alternative is defined using *sequences*. The *sequences* specify that the agent first has to go to the theatre to buy a ticket using the method "buyTicket", and afterwards to go to the restaurant to make a reservation for a table (a sequence is used here instead of a set because the agent needs the information when the theatre play ends to make the reservation).

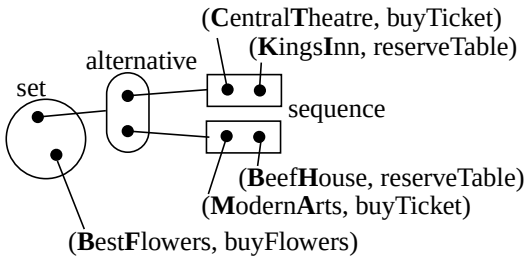


Figure 4.10a An itinerary...

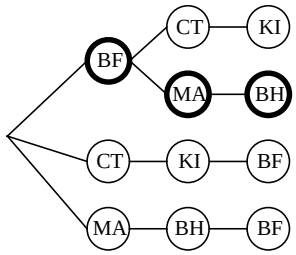


Figure 4.10b ... and the corresponding tree of possible paths

Given this itinerary, the system can decide which node has to be visited next provided there are alternatives. For the first step, it has the possibility to visit either one of the theatre nodes or the flower shop. The possibilities for the next step depend on the alternative chosen. Using the information given in the itinerary, a tree containing all possible paths of the agent can be constructed. The tree in Figure 4.10b shows all possible paths that can be taken by Pauls agent. The path, where the agent first orders the flowers, then buys a ticket in the ModernArts theatre and finally makes a reservation for a table at the BeefHouse is marked with bold circles.

The itinerary provides operations to query and to change its content. The query methods allow to navigate through the entries in the itinerary, to provide information about which nodes already have been visited and which nodes, according to the itinerary, may be visited next. The change methods allow to insert new entries and to delete entries in the part of the itinerary not yet processed. This allows the agent to gain an overview over the current state of its execution and to change the itinerary dynamically during its execution.

An Adapted "Exactly-Once" Property Definition

The adapted definition of the exactly-once property of mobile agents is based on the information contained in the agent´s itinerary and on the steps to be performed on the visited nodes.

Let $P=\{P_1,...,P_n\}$ be the set of all possible paths the agent may take for a given itinerary, let $L(P_i)$ be the number of nodes in path $P_i=[N_{i,1},N_{i,2},...,N_{i,L(Pi)}]$ and let $S_{i,j}$ be the step to be performed on the j-th node $N_{i,j}$ of path P_i ($1\leq i\leq n$, $1\leq j\leq L(P_i)$). Then the execution of an agent is defined to be exactly-once if

- only nodes $N_{i,1},...,N_{i,L(Pi)}$ belonging to one path $P_i\in P$ are visited,
- the agent executes step $S_{i,j}$ before step $S_{i,j+1}$, $1\leq j<L(P_i)$, and
- each step $S_{i,j}$ $1\leq j\leq L(P_i)$ is executed exactly once.

In the above scenario, a system providing the exactly-once property for mobile agents guarantees, that the agent visits the flower shop, only one of the two theatres and the restaurant associated with that theatre. The steps which have to be executed on these nodes are performed in one of the orders defined by the tree of possible paths in Figure 4.10b. Each of this steps is executed exactly once.

Types of Stage Nodes

A stage can be constructed using two different types of nodes, *regular nodes* and *exception handling nodes* (short: *exception nodes*). Let j be the node currently executing the agent, then *Nextj* defines the set of nodes that can be visited next according to the agent's itinerary. In the example depicted in Figure 4.10b the *Next* set associated with node *BF* includes nodes *MA* and *CT*. Let node *j* be the worker of stage *i*-1 then a node of stage *i* is called a *regular node* if it is member of *Nextj*. All other nodes of the stage are called *exception nodes*. In other words, regular nodes provide the services needed to perform the "regular" steps of an agent, while exception nodes are only expected to provide a runtime environment for agents. An agent is only moved to an exception node if in the stage no regular node is available due to failures. Each agent is supposed to provide a method *NoRegularNodeAvailable()*, which is initiated when the agent arrives on an exception node. As mentioned above, on regular nodes the method specified for the node in the itinerary is performed.

Stage Construction Algorithm

The algorithm described in this section aims at constructing a stage such that communication overhead is minimized during normal operation. The basic idea of the algorithm is to use as much as possible regular nodes to construct a stage and, if there is any freedom in the choice of nodes, to ensure that consecutive stages have as much nodes in common as possible. As an input this algorithm takes the number of stage nodes, say *n*, and the agent's itinerary. Assume node *w* is the worker of stage *i*. Then *w* performs the algorithm to determine the nodes of stage *i*+1, say S_{i+1}, together with the nodes' priorities.

Before describing the algorithm, we have to introduce some terminology. S_i is defined to be the nodes of stage *i* that are available from *w*'s point of view. The *Next* set is used as defined above,

i.e., $Next_w$ defines the set of nodes that potentially can follow *w* according to the agent's itinerary.

Case 1: In the simplest case, the cardinality of $Next_w$ ($|Next_w|$) equals *n*. In this case, S_{i+1} is equal to $Next_w$, which by definition includes regular nodes only. The way how priorities are assigned to these nodes will be described below.

Case 2: If the cardinality of $Next_w$ is bigger than *n*, then the resulting stage also contains only regular nodes. The choice of stage nodes is performed in two steps: In the first step, $S_{i+1}=S_i \cap Next_w$ is computed. If $|S_{i+1}|\geq n$, the priorities of the nodes in S_{i+1} are determined (see below) and the *n* nodes with the highest priorities remain in S_{i+1}. In this case, no second step is needed. If $|S_{i+1}|<n$, then $n-|S_{i+1}|$ nodes are selected from $Next_w \backslash S_i$ according to their priorities (see below). Subsequently, the final priorities of the nodes in S_{i+1} are calculated.

Case 3: If the cardinality of $Next_w$ is smaller than *n*, $m=n-|Next_w|$ exception nodes have to be chosen. Good candidates for this choice are the nodes in S_i, particularly *w*. By using these nodes, the number of code transfers for migrating the agent from stage *i* to stage *i*+1 can be reduced by $|S_i \cap S_{i+1}|$. In addition, using *w* as an exception node of stage *i*+1 reduces the number of data and execution state transfers and saves 4 messages during 2PC processing.

If $m\leq|S_i \backslash Next_w|$, *m* nodes - including *w* - are taken from $S_i \backslash Next_w$ as exception nodes according to their priorities. If $m>|S_i \backslash Next_w|$, then all nodes in $S_i \backslash Next_w$ are used as exception nodes. In order to select the yet missing exception nodes, future destinations specified in the itinerary can be taken into account. In sum, S_{i+1} includes $Next_w$ and a set of exception nodes selected as described above. Since $Next_w$ includes regular nodes, the exception nodes are assigned a lower priority than the ones in $Next_w$.

Priorities: To determine the priorities of the nodes, several possibilities exist. A simple approach is to randomly assign the priorities just ensuring unique priorities per stage. A more effective strategy is to exploit knowledge about node reliability. In this case, the priorities are assigned in accordance with the reliability, i.e., the higher the reliability, the higher the priority. Priorities between nodes with equal reliability can be chosen randomly. Obviously, with this heuristic more reliable nodes are preferred for agent execution.

A third strategy to determine the priorities is to take into account not only the next stage but also the ones following the next stage. Unfortunately, the nodes which can be visited in a stage depend on the worker of the previous stage, making the computations rather complex and time consuming. Therefore, this path hasn't been investigated further here.

The possible reductions of the protocol overhead gained by the presented algorithm are shown in Table 4.2. Assuming a number of stage nodes of *n*=3, the use of e.g. one node of the current stage as stage node of the next stage (regular or exception node) already reduces the number of code transports by one third.

The example itinerary shown in Figure 4.11 represents the optimal case for the stage construction algorithm. The first stage (*n*=3) contains nodes N_1 to N_3, ordered by their priority. For this first stage, only the code has to be transported to the nodes (3 code transports) and the transaction of putting the code onto the first stage nodes has to be committed (4*3 mes-

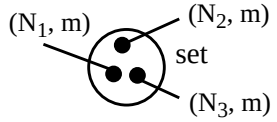


Figure 4.11 Simple itinerary

	Code Transports	State Transports	Messages for 2PC
$S_i \cap S_{i+1} = \{\}$	n	n	$4n$
$S_i \cap S_{i+1} \neq \{\}$	$n - \|S_i \cap S_{i+1}\|$	n	$4n$
$S_i \cap S_{i+1} \neq \{\}$ $w_i \in S_{i+1}$	$n - \|S_i \cap S_{i+1}\|$	n-1	$4(n$-1)

Table 4.2 Possible reductions of the overhead

sages). In the second stage, the worker of the first stage, (assume N_1), acts as exception node. Here, no code transports, only two state transports and 4*2 messages for the 2PC are necessary. In the third stage, N_1 and the worker of the second stage (assume N_2) act as exception nodes. The number of data transports is the same as in the last stage. Therefore, a total of 3 code transports (no overhead at all!), 4 state transports (overhead: 2 transports) and 28 messages for the 2PC are needed. Without the flexible definition of the itinerary and the optimization of the stage construction algorithm, 9 code transports, 9 state transports and 36 messages for 2PC would have been used.

In our example of Section , the first stage (n=3) contains the two ticket service shops and the flower shop (3 code transports + 4*3 messages). If the concierge agent is executed on the flower shop node first, the second stage consists of the two ticket service nodes (as regular nodes) and the flower shop node as exception node (two state transports and 4*2 messages). Then, the third stage consists of one of the restaurant nodes (depending on the ticket node used as worker in stage two) as regular node and the worker node of stage two and one other node of stage two as exception nodes (one code transport, two state transports and 4*2 messages). This results in a total of 4 code transports (overhead: one transport), 4 state transports (overhead: two transports) and 28 messages for 2PC. In this case, each stage contains as much as possible regular nodes. If, on the other side, the worker node of the first stage is one of the ticket service nodes, there can be an additional code transport if the flower shop is not contained in the second stage (the flower shop node is not in $Next_w$ in this case, which is rather an inadequacy of the itinerary concept than of the stage construction algorithm).

4.1.9 Related Work

In the field of mobile agents, only few research groups have considered aspects of transaction management and fault tolerance so far. In [MinsEA96] and [Sch97], a stage model similar to the one in this paper is proposed. However, the papers focus on a different aspect of fault-tolerance. Fault tolerance is achieved by processing the agent on each stage node (in parallel) and to send the migrating agent to all nodes of the following stage. Stage nodes perform voting on incoming agents to determine a majority of equal agents. Only an agent from this majority is processed further. Our approach assumes *alternative* services in a stage instead of replicated ones. Even in the case of failures, the agent is performed exactly once, e.g. in a car rental stage consisting of a Hertz, Avis and Budget server, the step “rent a car” is eventually performed once at one server. In [SK97], an agent- based transaction model is presented. Similar to our model, an agent

executes a transaction while moving from node to node. To prevent the blocking of agents due to long-lasting failures, the use of monitoring components is proposed. However, this paper purely concentrates on modelling aspects, protocols or algorithms are not given. [DalmEA98] introduces a mechanism based on a special system model. A site consists of several nodes interconnected by a reliable LAN. Agents within a site are monitored by a recoverable checkpoint manager (CM). The CM, which is assumed to be very reliable, is responsible to keep track of the agents and to restart the agents after a node has crashed and recovered. Unfortunately, this approach is very sensitive against CM crashes (no migration is possible during a CM crash) and long lasting node crashes (an agent cannot be restarted while a node is down). [VoKuMo97a] [VoKuMo97b] claim to realize exactly-once semantics for mobile agents by executing the migration between nodes in a distributed transaction. According to the description, the agents are not stored in stable storage. Considering this fact and the fact that there is no monitoring of agents, it can be assumed that agents are lost in case of a node crash and that agents are caught in case of a network partitioning. [JoReSc95] proposes an implementation of fault-tolerance using rear-guard agents, which are instantiated at migration time on the system the agents are leaving and which monitor the progress of the agents. Unfortunately, no details are given. Particularly, it is unclear how the blocking of agents on nodes can be prevented during a partitioning between the rear-guard and the agent. [BagcEA98] presents a mechanism to provide (non agent) applications with fault-tolerance using mobile agents.

In addition, there has been a lot of related work in the fields of transaction processing and fault-tolerant computing. The ConTract model [WR92][RSS97] also aims at the exactly-once property and similar to our approach only allows for forward recovery. A ConTract is defined by a script which is performed by a ConTract manager. A first prototype implementation, APRICOTS [Sch93], will be extended to allow the migration of scripts between ConTract managers, even in the case that the ConTract manager processing the script crashed, by using logging information written during the execution of the script to recover the state of the script on another node. However, there is currently no component which autonomously (and reliably) initiates the migration of a script to another ConTract manager if the ConTract manager executing the script crashes. Another approach is the use of process pairs [GR94] (also called hot backups [BE97]), where a primary process executes the program and sends checkpoint messages to a backup process. The backup process monitors the primary and takes over the execution using the latest checkpoint information received. In this approach, the communication between process pairs is assumed to be reliable. An extension to this approach are system pairs [GR94], where the processors can be geographically remote. This approach also cannot deal automatically with network partitioning. An approach based on replicated objects is presented in [BeedEA95]. Each replica of an object gets all messages sent to the object and executes all methods that are invoked. Only the primary sends messages (method calls, replies on method calls) to other objects. The system handles failures of primaries by selecting one of the replicas as primary. The communication system is assumed to be reliable and to offer an atomic broadcast. Despite the extraordinary overhead of executing all calls on an object on all its replicas, no byzantine errors are detected.

Voting algorithms are mainly used in the area of replicated data (e.g. [Thom79][Giffor79]) and mutual exclusion in distributed systems (e.g. [Maek85]). In our protocol, voting is used as a mutual exclusion mechanism preventing multiple stage nodes to commit.

4.1.10 Conclusion and Future Work

We have investigated how the exactly once property can be provided in mobile agent systems in a fault-tolerant way. We presented a protocol guaranteeing this property, while reducing the probability for agent blocking. Moreover, we proposed an architecture that allows to integrate the protocol in standard transactional technology. In other words, the proposed mechanism can be realized on top of conventional TP-monitors and transactional message queues. Currently, the protocol is under implementation in the Mole system [MOLE] and will be evaluated in terms of performance. Results are expected to be available in autumn 1998.

Future work will concentrate on enhancing our agent execution model. A straightforward extension is to allow the agent to access remote services resulting in distributed transactions which include arbitrary nodes. More sophisticated problems to solve are the communication with other mobile agents and the support of enhanced transaction models. One limitation of our current protocol is e.g. that, on system level, it allows for forward recovery only. In other words, if a user wants to abort an agent, the potentially required compensation operations are not automatically triggered on the system level. Instead, the logic to perform compensations must be provided in the agent by the agent programmer. This goes in line with the experiences made with today's workflow systems that many operations can only be compensated in an application-specific manner and often require the intervention of human users. However, some compensation can be done automatically, and we will investigate what concepts and protocols are needed to support compensation on the system level. Related problems also to be investigated are support of long-lasting actions including the recovery on other stage nodes without losing all work already done in the stage (e.g. by using safe points) and "atomic" actions over several nodes. We are confident that we can learn from the research conducted in the field of transaction models (e.g., Sagas [GMS87], open nested transaction [Wei91][WS91], etc.).

4.1.11 Appendix

The voting protocol described in the Section 4.1.5 is illustrated in this appendix using a pseudo-code notation. To keep the code as small as possible, several simplifications are performed. Most of the variables used in the code depend on the current *StageId* and should be read as *x*[*StageId*] instead of *x*. *VoterId* and *OrchId* are synonyms for the *NodeId* of the node the voter/orchestrator reside on. The orchestrator gets the information about a stage (*StageId*, *TransactionId*) from the entity performing the agent and is able to determine the *StageId* corresponding to the *TransactionId* (short: TId) and vice versa. All communication primitives are executed asynchronous. The tests using the TId to determine if votes are current are omitted.

Orchestrator

```
RECEIVE rm_prepare(TId){ // from TM
  YV=CYV=NV={}; // Vote Sets
  REPEAT
    SEND VOTE(StageId, TId, OrchId)
      TO StageVoters\(YV+CYW+NW);
    WAIT(sometime)
  UNTIL (majority achieved)
}
RECEIVE YES(StageId, TId, VoterId){
  YV = YV + VoterId;
  CheckCondYes(CYV, YV);
  IF YV contains majority THEN
    TA_State = Ready;
    SEND(rm_yes) TO TM;
}
RECEIVE NO(StageId, TId, VoterId){
  NV = NV + VoterId;
  IF NV contains majority THEN
    SEND(rm_no) TO TM;
    SEND(UN_VOTE(StageId, TId,
         OrchId)) TO
         All Voters in YW+CYW
}
```

```
RECEIVE COND_YES(StageId, TId
                 OrchSet, VoterId){
  CYW = CYW + (VoterId,OrchSet);
  CheckCondYes(CYV, YV);
  IF YV contains majority THEN
    TA_State = Ready;
    SEND(rm_yes) TO TM;
}
RECEIVE HIGHER_PRIO(StageId){
  IF TA_State==Unknown
    SEND(rm_no) TO TM;
    SEND(UN_VOTE(StageId, TId,
         OrchId)) TO
         All Voters in YW+CYW
    REPLY GAVE_UP;
  ELSE
    REPLY ALREADY_DONE;
}
PROCEDURE CheckCondYes(CYV, YV){
  FORALL (vid,orchSet) in CYW DO
    IF orchSet\YV == {} THEN
      CYV = CYV\(vid,orchSet)Set);
      YV = YV + vid;
}
RECEIVE GIVE_UP(StageId){
  SEND(rm_no) TO TM;
}
```

```
RECEIVE rm_commit(TId){ // from TM
  TA_State = Committed;
  SEND(rm_ack) TO TM;
  REPEAT
    SEND FORGET(StageId, OrchId)
      TO StageVoters;
    WAIT (sometime);
  UNTIL (all ACKs received)
  TA_State = Unknown;
}

RECEIVE rm_abort(TId){ // from TM
  TA_State = Unknown;
  SEND(UN_VOTE(StageId, TId,
       OrchId)) TO
       All Voters in YW+CYW
}
RECEIVE INQUIRY(StageId, VoterId){
  IF TA_State == Ready THEN
    YV = YV + VoterId;
  ELSIF TA_State == Committed THEN
    SEND FORGET(StageId, OrchId)
      to VoterId
  ELSE // Unknown
    IF (no active TA for StageId) THEN
      SEND UN_VOTE(StageId, Null)
        TO VoterId;
    ELSE
      // ignore
}
PROCEDURE RECOVERY(){
  FORALL TA with TA_state==Committed
  DO
    REPEAT
      SEND FORGET(StageId, OrchId)
        TO StageVoters;
      WAIT (sometime);
    UNTIL (all ACKs received)
    TA_State = Unknown;
}
```

Voter

```
RECEIVE VOTE(StageId,TId, OrchId){
  IF OrchSet=={} THEN
    OrchSet = OrchSet + OrchId;
    REPLY YES(StageId, TId, VoterId);
  ELSE
    N=node with
        highest priority in OrchSet;
    IF prio(OrchId) < prio(N) THEN
      REPLY NO(StageId, TId,
        VoterId);
    ELSE
      IF N not VoterId THEN
        REPLY COND_YES(StageId, TId,
          OrchSet, VoterId);
      ELSE
        SEND HIGHER_PRIO(StageId);
        RECEIVE ANSWER;
        IF ANSWER = GAVE_UP THEN
          IF OrchSet\{N} =={} THEN
            REPLY YES(StageId,
              TId, VoterId);
          ELSE
            REPLY COND_YES(StageId,
              TId, OrchSet\{N},
              VoterId);
        ELSE // ALREADY_DONE
          REPLY NO(StageId, TId,
            VoterId);
  IF voted YES or COND_YES THEN
    REPEAT
      WAIT(sometime);
      IF NOT RECEIVED (FORGET
        or UNVOTE) THEN
        SEND INQUIRY (StageId,
          VoterId) TO OrchId;
    UNTIL RECEIVED (FORGET or UNVOTE)
}
RECEIVE UN_VOTE(StageId,TId)
  FROM OrchId{
  OrchSet = OrchSet\{OrchId};
}
RECEIVE FORGET(StageId, OrchId){
  remove stage record and OrchSet;
  REPLY ACK(StageId);
  SEND GIVE_UP TO local orchestrator;
}
PROCEDURE RECOVERY(){
  FOR all OrchSets DO
    FOR all OrchIds in OrchSet DO
    REPEAT
      SEND INQUIRY (StageId,
        VoterId) TO OrchId;
    UNTIL RECEIVED (FORGET or UNVOTE)
}
```

4.2 Concepts for a Reliable and Scalable Agent Server

4.2.1 Requirements

Reliability: The agent system ensures, that the agents residing at the system don't get lost if some hardware or software components fail but that they get restarted in a well defined state. Especially in view of the use of mobile agent systems in the commercial environment, this requirement occupies a central role: nobody wants to lose money only because an agent carrying electronic cash gets lost.

Scalability: With the increasing use of mobile agents to access services, it is of first importance for service providers, that the agent execution environment scales to be able to flexibly react on an increased demand.

Communication and migration functionality should follow closely the functionality of the agent system Mole, which was developed at university of Stuttgart.
The agent system offers functionality to synchronously and asynchronously send messages. If a message can't be delivered to its recipient, it is ether queued and delivered at a later time or the sender gets an error message (depending of the specified failure semantics of the message). To deliver a message to an agent, the location calls the receiveMessage() method of the agent. Besides the message mechanism, the agent system offers a remote method invocation mechanism which enables the agent to invoke methods of other agents.
Additional to the direct addressing using unique agent names, it is able to use so-called *badges* for anonymous communication. Badges are tags (simple strings), which an agent may put on e.g. to announce that it offers a certain service.

4.2.2 Architecture of the Agent System

The agent system consists of several components. Figure 4.12 depicts these components and their interactions.

The basic idea of the agent system is to store all agent in a serialized form in a Tuxedo message queue. In the queue they are protected against software and hardware failures. To execute an agent ready for execution, the *Q-Manager* process gets the agent out of the message queue and sends the agent to a free server of the Tuxedo server class *Sublocation* using an asynchronous tpacall. The *Sublocation* server processes offer the *ExecuteAgent* function as its only service. This function takes a serialized agent as its parameter and executes this agent in a local (to the process) Java virtual machine. The *ExecuteAgent* function terminates if the agent terminates, if it migrates to another location or its execution state changes from "executable" to "blocked" (communication, sleep-functionality). In the last case, the *Q-Manager* re-inserts the serialized agent (and some additional state information) into the local message queue. All actions beginning from getting the agent out of the queue up to re-inserting the agent into the queue are performed within a transaction. In the case of an (hard-/soft-)failure of the Sublocation, the *Q-Manager* rolls the transaction back, if the *Q-Manager* fails, the transaction is rolled back automatically. If the transaction rolls back, the agent still resides in the input queue in its former state.

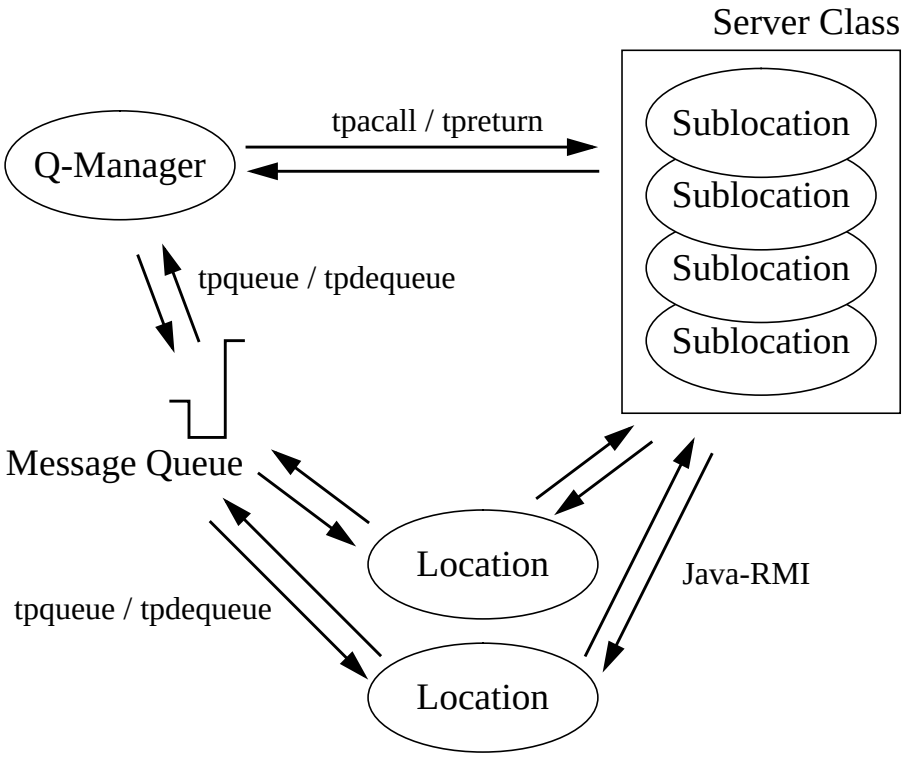


Figure 4.12 Architecture of the agent system

Locations are places where agent are instantiated, to which agents can migrate and where agents "live" logically. Locations manage the agents residing at this location. One of their tasks is to make sure that blocked agents residing in the message queue are executed again if an communication request (message, remote method invocation) concerning this agent arrives.

For performance reasons, the real communication takes place directly between the involved agents. Hereby, the agents are supported by their sublocation. A sublocation is some kind of runtime environment which among others provides the agent with the ability to communicate and to migrate. A sublocation is launched each time the ExecuteAgent service is invoked. The sublocation registers the agent at the local RMI registry to be able to receive messages and remote method invocations for the agent. Afterwards, the agent is registered at its location by passing the RMI-URL under which the agent can be reached. Before the ExecuteAgent service terminates, the agent is removed at the local RMI registry and its location.

4.2.3 Lifecycle of an Agent

The state diagram in Figure 4.13 shows the life cycle of an agent, which will be explained in detail in the next paragraphs.

An agent can be started in two different way. On the one hand, an agent can be started from the command line of a location using the new command (syntax: new <agent class> [<tag><parameter>[, <tag> <parameter>[,...]]];). On the other hand, an agent can be launched by an other agent using the createAgent() method provided by the location. After the creation and the ini-

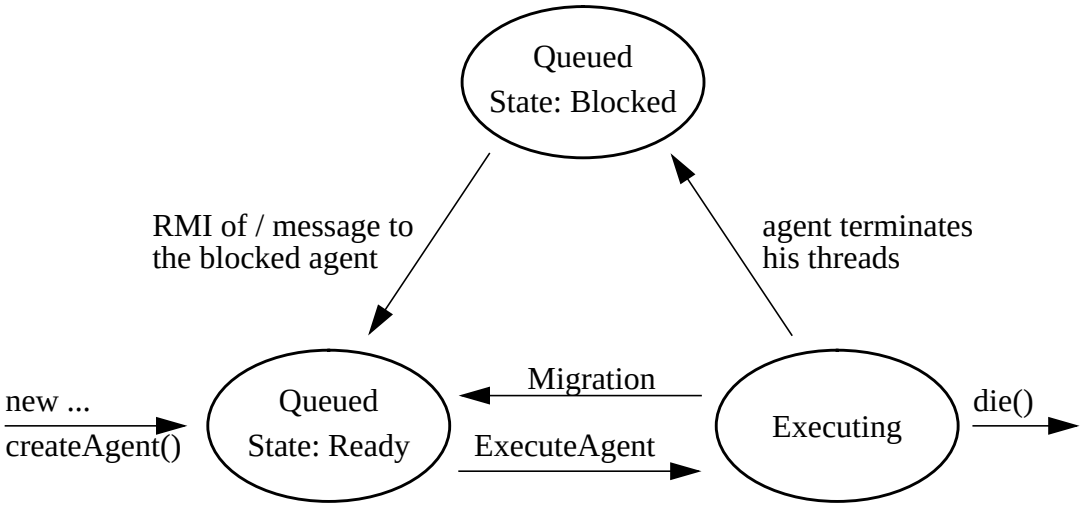


Figure 4.13 Life cycle of an agent

tialization of the agent by the location, the agent including the information that the agent is ready to run is put into the message queue. The location additionally stores some other informations about the agent in a list containing all agents residing on this location. As soon as a server of the class ExecuteAgent is available, the Q-Manager starts a new transaction, reads the agent from the message queue and invokes its execution using the asynchronous call (tpacall) of the ExecuteAgent service. The ExecuteAgent service now starts a new sublocation. This sublocation registers itself (using the agent's name) at the local RMI registry to be able to receive (and to deal with) messages and remote methods invocations for the agent. Then, the sublocation registers the agent at its location using the registerAgent() method of the location. The location updates its informations about the agent (e.g. "agent xy now being performed in service process z") and returns state information to the sublocation. This state information tells the sublocation what to do with the agent. If the agent is newly started or if it is just migrated to the location, the start() method of the agent has to be invoked. If the agent stayed in the queue while being blocked (sleeping, waiting for communication), according steps have to be taken (see below).

The start() method of an agent implements the desired functionality of the agent. During the execution of start(), the agent may communicate to other agents (messages, remote method invocations), use local services and invoke the migration to other locations. It is important to note that, because the agent always starts its execution on a location with the start() method, the agent itself always has to know what to do on the location it just arrived (e.g. by using some information stored in the agents data).

If the agent intends to terminate itself, it calls the die() method. This method removes the agent at the location and the local RMI registry. Then, the execution of the ExecuteAgent service is terminated using the tpreturn call. The Q-Manager, which periodically checks if a service request returned (using tpgetrply), now commits the transaction (which was started before the agent was read from the queue).

If, during the execution, an agent has no more active threads (e.g. if it sleeps or waits for communication requests), the agent can be put back into the message queue to enable other agents to be executed. In this case, the sublocation informs the location on the agent's state (waiting

for communication, sleeping) and remove the agent (more exact: itself) from the local RMI registry. Then, the execution of the ExecuteAgent service is terminated using the tpreturn call, informing the Q-Manager to put the agent (and its state “blocked”) into the message queue and to commit the transaction.

If the reason that the agent is blocked doesn’t hold any longer (communication request or time to sleep is exceeded), the location has to reactivate the agent. Therefore, it gets the agent from the message queue and puts the agent back in the queue with the information that the agent is ready to be executed. As soon as a server from the ExecuteAgent class is available, the Q-Manager invokes the agent execution as described above. If the sublocation executing the agent now registers itself at the location, the location provides the information why the agent has been activated (message, remote method invocation, time to sleep exceeded, combinations are possible) enabling the sublocation to invoke the required functionalities (receive the message, call the desired method or activate the agent after its sleep).

4.2.4 Messages

The sublocation offers two different methods for message transmission for the agent. The synchmessage() method transfers messages synchronously using the thread in which the method has been called. The message() method transfers messages asynchronously by starting a new thread in which synchmessage() is executed. This enables the agent to continue its execution while the message is being transferred.

Both methods require as a parameter an error sematics specification defining their behaviour in the case that the message can’t be delivered to the receiver agent. Currently, three different possibilities exist. In the first case, the message is only discarded if the receiver agent does not reside at the specified location, no error message is generated. In the second case, the message is discarded and an error message is sent to the sender of the message. In the third case, the message is stored at the specified receiver location and is delivered when the receiver agent arrives at the location.

Other parameters of the two methods are the receiver of the message and, of course, the content of the message. The receiver of the message is specified by providing the receiver agent’s unique agent name and the location where the receiver agent resides. The location has to be specified because there is currently no service to map an agent’s name to its current location.

In the following paragraphs, a short description of the synchmessage() method is given. If the synchmessage() is invoked, the sublocation of the sender first invokes the getAgentRMIURL() method of the sender location to get the RMI-URL of the receivers sublocation. If the sender and the receiver reside at different location, the senders location passes the method invocation to the receiver location. If the receiver is currently being executed by the ExecuteAgent service, getAgentRMIURL() immediately returns the existing RMI-URL of the receiver agent’s sublocation. If the receiver agent currently stays in the message queue, its location reactivates the agent by reading it out of the message queue and re-inserting it with the information that the agent is now ready for execution. Now the thread executing getAgentRMIURL() is suspended until a sublocation registers the receiver agent with the location via registerAgent(). Now, getAgentRMIURL() is able to return the agent’s RMI-URL.

The sending sublocation is now able to get a reference to the receiver's sublocation using the RMI lookup (with the RMI-URL received by getAgentRMIURL()). This reference is used to invoke the deliverMessage() method of the receivers sublocation. This method now invokes the best-fitting receiveMessage() method of the agent using the java reflection API (the agent may contain several methods having different message types (classes) as (its only) parameter).

Figure 4.14 shows the synchronous message transfer in the case that sender and receiver agent both reside at the same location (black bars symbolize executing threads).

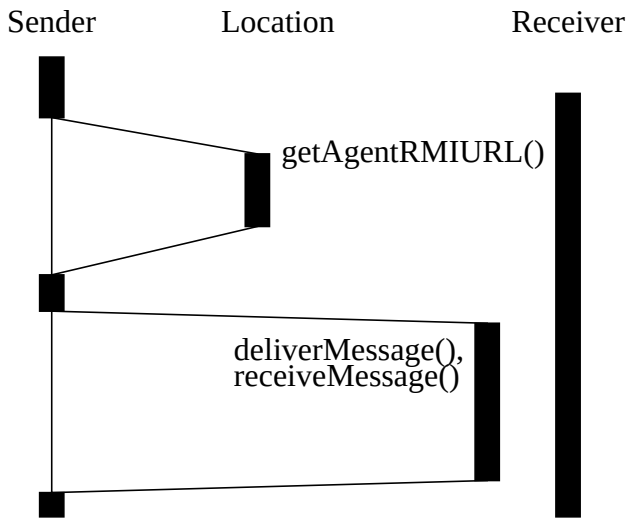


Figure 4.14 The synchmessage() method

In general, two communicating agents send more than one message in both directions. To enhance the efficiency of the communication by avoiding to resolve the receiver agents sublocation (by calling getAgentRMIURL() and resolving the reference by RMI lookup) for each communication, a cache is used. This cache maps agent names to the last known sublocation reference. At each invocation of synchmessage(), the cache is searched for an entry for the receiver agent. If the cache contains an entry, deliverMessage() is invoked with this reference as a parameter. If the reference contained in the cache is obsolete (e.g. because the agent stays in the message queue), an exception occurs. In this case as well as the case that there is no cache entry for the agent, getAgentRMIURL() and a RMI lookup are invoked as described above and the obtained reference is stored in the cache.

The receiver agents may not only be addressed using their (unique) agent name, but it is also possible to send messages to anonymous recipients by using a badge. An agent may pin on a badge (containing a string) which can then be used to address the agent (instead of the agents name). If several agents on a location have pinned on the same badge, messages being addressed to agents wearing this badge are distributed "round-robin" to the agents wearing this badge. Because agents are able to remove a badge they have pinned on and because badges can be used to realize a rudimentary load balancing between several agents, agent sublocations which are resolved from a badge are not stored in the cache.

4.2.5 Remote Method Invocation

To invoke a method offered by another agent, the sublocation offers a call()-method to its agent. This method takes as parameters the name and the parameters of the method which should be called and the name (or badge) and location of the agent offering the method.

The execution of the call() method is very similar to the execution of synchmessage(). By means of getAgentRMIURL() and RMI lookup, the reference of the sublocation of the receiver agent is resolved. This reference is then used to invoke the dispatchRPC() method of the receiver agent. This method uses the Java reflection API to invoke the desired method.

Figure x may also be used to show the execution of a remote method invocation (substitute synchmessage() by call() and deliverMessage()/receiveMessage() by dispatchRPC())

4.2.6 Migration

If an agent wants to continue its execution on another location, it calls the migrateTo(destinationLocation) method of its sublocation. This method first suspends all active threads of the agent (of course all but the thread executing the migrateTo()). Then, the RMI-URL of and the reference to the destination location is determined. This reference is used to invoke the handleMigration() method of the destination location, passing the serialized agent as parameter. The handleMigration() method inserts the agent in its message queue.

If the agent is passed successfully to the other location, the “old” sublocation deletes its entry in the RMI registry and its location. Then, the ExecuteAgent service terminates and the Q-Manager commits the affiliated transaction. If the agent cannot be passed successfully to the destination location, the suspended threads of the agent are reactivated

4.2.7 Current State

The agent server is currently under implementation on a Tandem Himalaya under OSS. First results are expected in June 1998.

4.3 Bibliography

[AA88] Abu-Amara, H.H. 1988. “Fault-tolerant distributed algorithm for election in complete networks.” IEEE Transactions on Computers, 37(4). 449-453

[AgrAbb91] Agrawal, D.; El Abbadi, A.: “An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion”. In: ACM Transactions on Computer Systems, Vol. 9, No. 1. February 1991, pp. 1-20

[BauEA98] Baumann, J.; Hohl, F.; Rothermel, K.; Straßer, M.: “Mole - Concepts of a Mobile Agent System.” accepted for “WWW Journal, Special issue on Applications and Techniques of Web Agents”, Baltzer Science Publishers, 1998.

[BE97] Bernstein, P.; Newcomer, E.: “Principles of Transaction Processing”. Morgan Kaufmann Publishers Inc, 1997

[BeedEA95] Beedubail, G., A. Karmarkar, A. Gurijala, W. Marti, and Udo Pooch. 1995. „Fault Tolerant Objects in Distributed Systems Using Hot Replication“. Technical Report

TR_95-023. Department of Computer Science. Texas A&M University
[BeHaGo87] Bernstein, P.; Hadzilacos, V.; Goodman, N.: “Concurrency Control and Recovery in Databas Systems.” Addison-Wesley Publishing Company. 1987
[Blak95] Blakeley, B.: “Messaging and Queuing Using the MQI”, McGraw-Hill series on computer communications, 1995
[ChaCha97] Chang, Y., and Chang, Y.: “A Fault-Tolerant Triangular Mesh Protocol for Distributed Mutual Exclusion.” In: Operating Systems Review, Vol. 31, No. 3, July 1997. ACM Press. pp 29-44
[Giffor79] Gifford, D.K. 1979. “Weighted Voting for Replicated Data.” Proc. 7th Symp. on Operating System Principles 1979 (SOSP’’79). ACM, New York. pp. 150-162
[GM] General Magic: “Agent Technology”, http://www.genmagic.com/agents/
[GM82] Garcia-Molina, H.: “Elections in a Distributed Computing System”. IEEE Transactions on Computers, Vol. C-31(1), 1982
[GMS87] Garcia-Molina, H.; Salem, K.: “Sagas”. In: Proc. ACM Conf. on Management of Data, pp. 249-259, 1987
[GR94] Gray, J.; Reuter, A.: “Transaction Processing - Concepts and Techniques”. Morgan Kaufmann Publishers Inc, 1994
[GV97] Ghezzi, C.; Vigna, G.: “Mobile Code Paradigms and Technologies: A Case Study”. In: Mobile Agents, Proc. 1st Int. Workshop, MA’97. Springer, 1997
[HR93] Haerder, T.; Reuter, A.: “Principles of Transaction-Oriented Database Recovery.”. ACM Computing Surveys, 15(4), 1993
[HuGr71] Hughes, A.; Grawoig, D.: “Statistics: A Foundation for Analysis.” Addison-Wesley Publishing Company, 1971.
[IyKaBaA97] Iyer, R.K.; Kalbarczyk, Z.; Bagchi, S.: “Chameleon: A Software Infrastructure and Testbed for Reliable High-Speed Networked Computing”. UIUC (University of Illinois at Urbana-Champaign) Technical Report No.UILU-ENG-97-2218, July 97.
[Jal94] Jalote, P.: “Fault Tolerance in Distributed Systems”. Prentice Hall Inc., 1994
[JoReSc95] Johansen, D.; van Renesse, R.; Schneider, F.B.: “Operating system support for mobile agents.” Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems. IEEE. 1995
[Lam81] Lampson, B.: “Atomic Transactions”. In: Lampson, B. et al (eds): “Distributed Systems - Architecture and Implementation”, Springer-Verlag, 1981
[LO97] Lange, D.; Oshima, M.: “Java Agent API: Programming and Deploying Aglets with Java”. To be published by Addison-Wesley, Fall 1997; a working draft, “Programming Mobile Agents in Java, is available at http://www.trl.ibm.co.jp/aglets/whitepaper.htm.
[Maek85] Maekawa, M.: “A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems.” In: ACM Transactions on Computer Systems, Vol. 3, No. 2, May 1985, pp 145-159
[MinsEA96] Minsky, Y.; van Renesse, R.; Schneider, F.B.; Stoller, S.D.: “Cryptographic Support for Fault-Tolerant Distributed Computing.” In:Proceedings of the Seventh ACM SIGOPS European Workshop. 1996. pp 109-114.
[MNHT89] Masuzawa, T., N. Nishikawa, K. Hagihara, and N. Tokura. 1989. “Optimal fault-tolerant distributed algorithms for election in complete networks with a global sense of direction.” In Proceedings of the 3rd Int’l Workshop on Distributed Algorithms.
[Mole] Project Mole, http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html
[OMG96] Object Management Group: “CORBA 2.0 specification”, ptc/96-03-04, 1996

[PS97] Peine, H.; Stolpmann, T.: “The architecture of the Ara platform for mobile agents.”. In: Mobile Agents, Proc. 1st Int. Workshop, MA’97. Springer, 1997

[RoSt98] Rothermel, K.; Straßer, M.:”A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents”. Accepted for 17th IEEE Symposium on Reliable Distributed Systems. 1998

[RSS97] Reuter, A.; Schneider, K.; Schwenkreis, F.: “ConTracts Revisited”. In: S. Jajodia and L. Kerschberg (ed.): Advanced Transaction Models and Architectures (ATMA), Kluwer Verlag, 1997

[SBH96] Straßer, M.; Baumann, J.; Hohl, F.: “Mole - A Java Based Mobile Agent System”. In: Mühlhäuser, M.: “Special Issues in Object-Oriented Programming”, Workshop Reader ECOOP’96, p327-334, dpunkt.verlag, 1996. pp 327-334

[Sch93] Schwenkreis, F.: “APRICOTS - Management of the Control Flow and the Communication System”. In Proc. of the 12th IEEE Symposium on Reliable Distributed Systems, Princeton, October 1993

[Sch97] Schneider, F.: “Towards Fault-tolerant and Secure Agentry”. In: Proc. 11th Int. Workshop on Distributed Algorithms, 1997

[Singh96] Singh, G. 1996. “Leader Election in the Presence of Link Failures.” IEEE Transact. on Parallel and Distributed Computing, 7(3)

[Singh97] Singh, G. 1997. “Leader Election in Complete Networks.” SIAM Journal on Computing, 26(3). June 1997

[SK97] Morais de Assis Silva, F.; Krause, S.:”A Distributed Transaction Model based on Mobile Agents”. In: Mobile Agents, Proc. 1st Int. Workshop, MA’97. Springer, 1997

[Spe82] Spector, A.: “Performing remote operations efficiently on a local computer network”, Communications ACM, vol. 25, pp 246-260, Apr. 1982

[StRoMa98] Straßer, M.; Rothermel, K.; Maihöfer, C.: “Providing Reliable Agents for Electronic Commerce.” In: “Trends in Distributed Systems for Electronic Commerce”, International IFIP/GI Working Conference (TREC’98), Hamburg, Germany, June 1998. Lamersdorf, Winfried; Merz, Michael (Eds.), Lecture Notes in Computer Science 1402, Springer Verlag, Berlin, Pages 241-253. 1988

[Thom79] Thomas, R.H. 1979. “A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases.” ACM Transactions on Database Systems, Vol. 4, No. 2., June 79.pp. 180-209.

[VoKuMo97a] Vogler, H.; Kunkelmann, T.; Moschgath, M.L.: “Distributed Transaction Processing as a Reliability Concept for Mobile Agents.” In: 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS’97). IEEE Computer Society. 1997. ISBN 0-8186-8153-5.

[VoKuMo97b] Vogler, H.; Kunkelmann, T.; Moschgath, M.L.: “An Approach for Mobile Agent Security and Fault Tolerance using Distributed Transactions.” In; Proc. 1997 Int’l Conference on Parallel and Distributed Systems (ICPADS’97). IEEE Computer Society. 1997 ISBN 0-8186-8227-2

[Wei91] Weikum, G.: “Principles and realization strategies of multi-level transaction management”. ACM Transactions on Database Systems, 16(1): pp. 132-180, March 1991

[WoEA97] Wong, D.; Paciorek, N.; Walsh, T.; DiCelie, J.; Young, M.; Peet, B.: ”Concordia: An Infrastructure for Collaborating Mobile Agents.” In: Rothermel, K.; Popescu-Zeletin, R. (eds.): “Mobile Agents. First International Workshop MA ‘97.” Lecture Notes in

Computer Science, Vol. 1219, Springer. 1997, pp. 86-97.
[WR92] Wächter, H.; Reuter, A.: “The ConTract Model”. In: A. Elmagarmid (ed), Database Transaction Models for Advanced Applications, Morgan-Kaufmann, 1992
[WS91] Weikum, G.; Scheck, H.: “Multi-level transactions and open nested transactions”. IEEE Data Engineering Bulletin, March 1991
[X/O91] X/Open DTP:“X/Open Common Application Environment”, “Distributed Transaction Processing:Reference Model”, “Distributed Transaction Processing: The XA Specification” Reading, Berkshire, England: X/open Ltd, 1991

5 WP 2.4: Developed Concepts and Implementation

Version 3.0 of Mole has been strongly revised and several requests and proposals from users of the earlier versions of Mole were integrated into the new release. In particular Mole supports communication between agent groups and concept of sessions. The infrastructure of Mole includes a resource manager, a directory service and a global naming scheme for agents. In order to support the design of agents, a graphical agent monitor allows to visualize the system behaviour as a whole or of a single agent in particular. Mole further provides a thread management unit to overcome some shortcomings of the Java virtual machine. Mole provides a simple means for installation and configuration of the system. This chapter summarizes a paper that will be published at the Middleware'98 conference by Baumann, Hohl, Rothermel, Schwehm and Straßer (1998).

5.1 Mole 3.0: A Middleware for Java-Based Mobile Software Agents

5.1.1 Introduction

Mobile agents are a new programming model for distributed systems. Generally a mobile agent is a process that can act within a distributed system on behalf of its user. In particular such processes must be able to move freely from node to node in a distributed system and to continue processing asynchronously even if its user is (temporarily) not connected with the system any more. To allow such a functionality, each computing node of the distributed system must provide a suitable infrastructure. Such a platform for mobile agents consists basically of a virtual machine (engine) that must run on each participating node of the distributed system. The engine manages two types of objects: places and agents. A place is an object that provides an infrastructure for executing agents. An agent is a process that can occupy a place and that can communicate with other agents. Stationary agents can provide services and system resources to other agents. A mobile agent can actively move from one place to another (agent migration) and can access or provide services by communication with other agents. Applications of mobile agents can be found among others in the areas of network management, electronic commerce and mobile computing.

The idea to send machine independent executable messages via a network can be traced back to the very beginning of the Internet, where the Decode-Encode-Language (DEL) was considered to run interactive programs on remote consoles of a networked system (RULIFSON 1969). Later the idea emerged independently in the area of radio network communication, where the SOFTNET project used Forth-messages to transmit data as well as to reprogram the underlying network (ZANDER 1981). Another early approach was the Network Command Language by FALCONE (1987). In the nineties the term 'messengers' was used by TSCHUDIN (1993) to denote active messages programmed in his Postscript-like language M0. The term 'mobile agent' was coined in a white paper by General Magic Inc. (1994, republished by WHITE 1997a, 1997b). General Magic's Telescript language was specifically designed for mobile agent programming and already included most of the concepts of later mobile agent systems, but it was dropped when it became clear that it could not compete with Java as a commercial product. In the sequel several mobile agent systems have been developed in the research community. The research sys-

tems were based on such diverse programming languages like untyped scripting languages, e.g. Agent Tcl, an extension of Tcl by GRAY (1995, 1996), or strongly typed functional languages like MAP based on Scheme by PERRET and DUDA (1996), but the mainstream of the systems today are based on Java. Examples are Mole by STRAßER, BAUMANN and HOHL (1996) or Aglets by the IBM AGLETS WORKBENCH TEAM (1997). Other approaches use a language independent approach like TACOMA by JOHANSEN, VAN RENESSE and SCHNEIDER (1994, 1995) or Ara by PEINE and STOLPMANN (1997). Recently, several companies and research groups have presented MASIF, a proposal for the standardisation of mobile agent systems (MILOJICIC *et al.* (1998)).

This paper presents Version 3.0 of Mole, one of the first Java-based mobile agent systems. The paper proceeds with an overview of the Mole system in Section 2. Then basic concepts of the Mole system like agent migration in Section 3 and agent communication in Section 4 are presented. Section 5 continues with a description of the agent infrastructure provided by Mole and Section 6 introduces the graphical agent monitor of the system. Implementational issues are discussed in Section 7 and the installation procedure is sketched in Section 8.

5.2 Mole System Overview

Our model of an agent-based system - as various 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. In our model (see Figure 5.1)., agents may be multi-threaded entities, whose state and code is transferred to the new place when agent migration takes place. 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 at a single node of the underlying network, but multiple places may be running 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 underly-

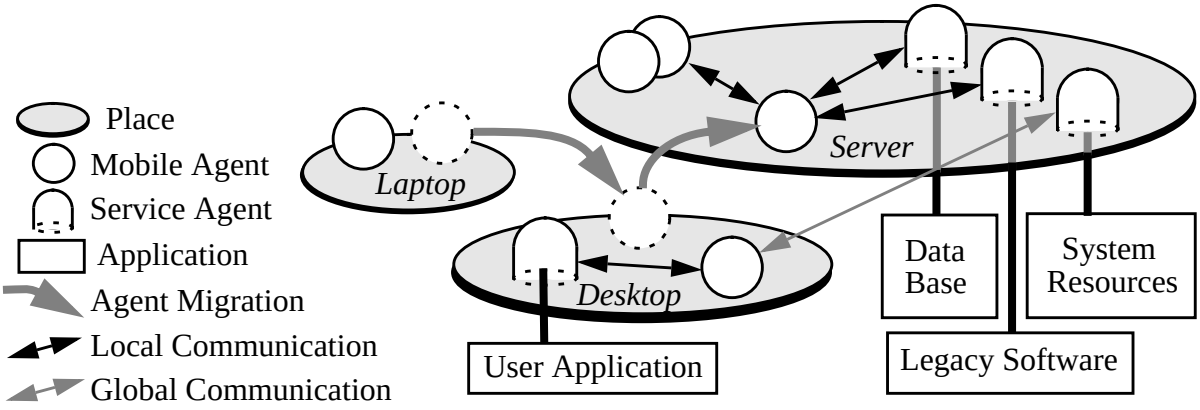


Figure 5.1 Mole System Overview

ing system. If a system is connected to the network all the time (exept for network failures and system crashes), a place on this system is called connected. If a system is only temporarily connected to the network, e.g. a user's PDA (Personal Digital Assistant), the place is called associatedAgent Lifecycle and Agent Mobility

5.2.1 Lifecycle of an Agent

After an agent is created, its initialization routine `init()` is processed (see Figure 5.2). The arguments given to the agent at creation time are passed to this routine. The programmer can set up the internal state and initialize the agent attributes. At this time the agent is still outside any place. Now the agent system injects the agent into the system as if it had just arrived after a migration. First the agent is made known to the place, but other agents are not yet allowed to communicate with it. The `prepare()`-method is called, allowing the agent to do its place-specific setup, e.g. identifying local services. Finally the agent is started by calling the `start()`-method. After that normal processing takes place, the agent can start its own threads, use local services, and can communicate locally or remotely. The agent stays on the place until it decides to either migrate or to terminate.

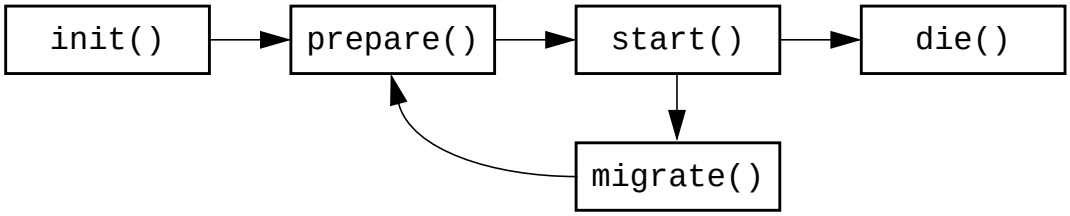


Figure 5.2 Lifecycle of a Mobile Agent

If the agent wants to migrate it calls the method `migrateTo()` (described in more detail in the next section). The system suspends the agent's threads, serializes the agent and sends it to the target place. If the target place accepts the agent, the agent is injected into the system and started again via the methods `prepare()` and `start()`. Now the target place sends an acknowledgement back to the source place which removes the suspended agent from the system. If the target place does not accept the agent, an error message is sent back and the agent resumes its work on the original place. It receives an exception as the result of the failed migration, and can react e.g. by trying to migrate to another place. If the agent has reached the end of its life, it calls the method `die()`. The system now stops all threads of the agent, removes it from the place, and deletes the agent.

Additionally the system supports periodic operation. This provides a simple mechanism for the programmer to implement recurring tasks, e.g. checking a database for changes if no trigger mechanism exists. If an agent implements the interface `Periodical`, then the system, in addition to calling the `init()`-method, executes the method `heartbeatInit()`. Now as soon as the `start()`-method has been called for the first time, the system begins executing a method called `heartbeat()` in regular intervals. This continues until the `die()`-method is called.

5.2.2 Agent Migration

The concept of a mobile agent supports process mobility, i.e., program executions may migrate from node to node of a computer network. Obviously, for migrating agents not only code but also the state information of the agent has to be transferred to the destination. An agent's state is subdivided into data state and execution state. While the first includes the agent's global and instance variables, the latter comprises the local variables and the active threads. According to GHEZZI and VIGNA (1997) two types of agent migration can be distinguished: weak migration and strong migration. With strong migration, the underlying system captures the entire agent state (consisting of data *and* execution state) and transfers it together with the code to the next location. Once the agent is received at its new location, its state is automatically restored. From a programmer's perspective, this scheme is very attractive since capturing, transfer and restoration of the complete agent state is done transparently by the underlying system. On the other hand, providing this degree of transparency in heterogeneous environments at least requires a global model of agent state as well as a transfer syntax for this information. Moreover, an agent system must provide functions to externalize and internalize agent state. Since the complete agent state (including data and execution state) can be large - in particular for multi-threaded agents - strong migration might be a very time-consuming and expensive operation. These difficulties have led to the development of the so-called weak migration scheme, where only data state information is transferred. The size of the transferred state information can be limited even more by letting the programmer select the variables making up the agent state. As a consequence, the programmer is responsible for encoding the agent's relevant execution states in the program variables. Moreover, the programmer must specify a start method that decides on the basis of the encoded state information where to continue execution after migration. While this method may substantially reduce the amount of state to be communicated, it puts additional burden on the programmer and makes agent programs more complex. The difference between weak and strong migration is a change in semantics, but not in expressive power. One of the design goals of Mole is the ability to run on every Java Virtual Machine (VM). A normal Java VM does not support capturing the state of a thread, which would be a prerequisite for capturing the execution state. Thus our decision was to choose the changed semantics of weak migration and with it the ability to run Mole on unchanged Java interpreters. This includes that, while agents in Mole can be multithreaded, after a migration only one thread is started. If more threads are necessary the agent has to start them explicitly.

Weak migration in Mole is implemented by using a part of the Remote Method Invocation package RMI, the object serialization provided as part of Java 1.1. After an agent thread calls the `migrateTo()`-method, all threads belonging to the agent are suspended (not stopped). No new messages and calls (RPC) to the agent are accepted. After all pending messages to the agent have been delivered, the agent is removed from the list of active agents. Now the agent is serialized using the object serialization. The object serialization computes the transitive closure of all objects belonging to the agent (ignoring transient objects and threads), and creates a system-independent representation of the agent. This serialized version of the agent is sent to the target place that reinstantiates the agent. If any of the Java classes needed are not available locally, the target place requests these classes either from a code server (HOHL *et al.*, 1997), or from the source place. Now the agent is reinstantiated. One new agent thread is started. First the `prepare()`-method is called to initialize the place-specific attributes. Then this thread begins its work at the `start()`-method. As soon as the thread assumes control of the agent, a success message is sent back to the source place. The source place now terminates all threads pertaining

to the agent and removes it from the system at any stage of the migration an error occurs, the migration is stopped and the agent threads at the source place are resumed. The control flow continues after the `migrateTo()` statement, where error handling can be implemented (an exception is thrown in the case of failure).

5.3 Agent Communication and the Session Concept

As will be seen below, a session between agents can be established only if the agents can identify each other. In our model, there are basically two ways how agents can be identified, the unique agent identifiers - comparable to object identifiers - and the so-called badges. Agent identifiers are well-suited for identifying service agents, as long as there exists a directory system, that maps user-defined service names to service agent identifiers. Note, however, that the directory service is not part of our base system, i.e., we clearly separate the mechanism for identifying services from the one for finding services. As a consequence, different naming schemes and directory systems can be used on top of this system. The directory service provided by Mole is described in Section 5.2.

5.3.1 Badges

In the case of mobile agents the concept of agent identifiers is not always sufficient. Assume for example, that an agent wants to meet some other agent participating in the same task at a given place. If only agent identifiers were available, both agents would have to know each others identifiers. Actually, for identification it would be sufficient to say “At place XYZ I want to meet an agent participating in task ABC”. This type of identification is supported by the concept of badges. A badge is an application-generated identifier, such as “task ABC”, which agents can “pin on” and “pin off”. This badge does not necessarily have to be unique, it simply represents a role of an agent at a given time. As long as the agent provides the functionality associated with this role, it wears the badge. An agent may have several badges pinned on at the same time. Badges may be copied and passed on from agent to agent, and hence multiple agents can wear the same badge. For example, all agents participating in a subtask may wear a badge for the subtask and another one for the overall task. The agent that carries the result of the subtask may have an additional badge saying „CarryResult“. Using badges, an agent is identified by a (place identifier, badge predicate)-tuple, which identifies all agents fulfilling the badge predicate at the place identified by the place identifier. A badge predicate is a logical expression, such as (“task ABC” AND (“CarryResult” OR “Coordinator”)). Obviously, this is a very flexible naming scheme, which allows to assign any number of application-specific names to agents. To change the name assignments two functions are provided, `PinOnBadge(badge)` and `PinOffBadge(badge)`.

5.3.2 Sessions

A session defines a communication relationship between a pair of agents. Agents that want to communicate with each other, must establish a session before the actual communication can be started. After session setup, the agents can interact by remote method invocation or by message passing. When all information has been communicated, the session is terminated. Sessions have

the following characteristics: Sessions may be intra-place as well as inter-place communication relationships, i.e., two agents participating in a session are not required to reside at the same place. In order to preserve the autonomy of agents, each session peer must explicitly agree to participate in the session. Further, an agent may unilaterally terminate the sessions it is involved in at any point in time. Consequently, agents cannot be trapped in sessions. While an agent is involved in a session, it is not supposed to move to another place. However, if it decides to move anyway, the session is terminated implicitly. The main reason for this property of sessions is to simplify the underlying communication mechanism, e.g., to avoid the need for message forwarding.

The question may arise, why sessions are needed at all. There are basically two reasons: Firstly, the concept of a session can be used to synchronize agents that want to meet for cooperation. Note that the first property stated above allows agents to meet even if they stay on different places. The concept of a session is introduced to allow agents to specify which other agents they are interested to meet at which places. Furthermore, it allows agents to wait until the desired cooperation partner arrives at the place and indicates its willingness to participate. Secondly, we want to support both "stateless" and "stateful" interactions. In contrast to the first, the latter maintain state information for a sequence of requests. Obviously, if they encapsulate "stateful" servers, service agents have to be "stateful" also. A prerequisite for building "stateful" entities are explicit communication relationships, such as sessions.

In order to establish sessions, two methods are provided, `PassiveSetUp()` and `ActiveSetUp()` (see Figure 5.3). The first operation is non-blocking and is used by agents to express that they are willing to participate in a session. In contrast, `ActiveSetUp()` is used to issue a synchronous setup request, i.e., the caller is blocked until either the session is successfully established or a timeout occurs.

```
void PassiveSetUp(PeerQualifier , PlaceId)
SessionObject ActiveSetUp(PeerQualifier, PlaceId, Timeout)
boolean SetUp(SessionObject)
void SessionObject.Terminate()
```

Figure 5.3 Session Methods

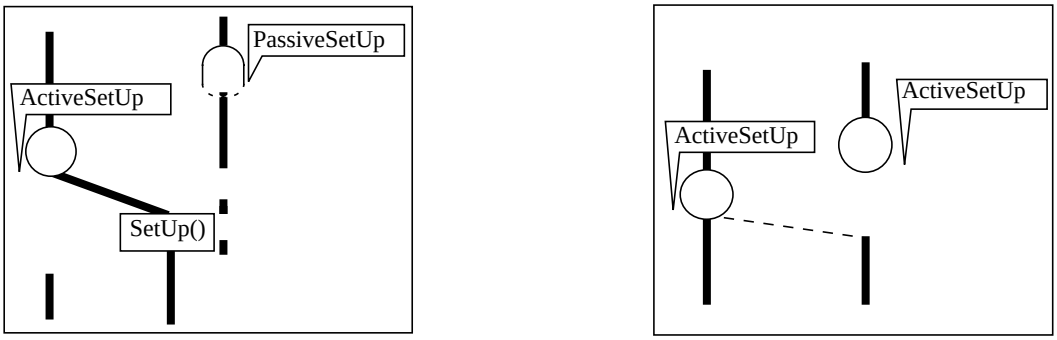
If `ActiveSetUp()` succeeds, it returns the reference of the newly created session object to the caller. Input parameter `PlaceId` identifies the place, where the desired session peer is expected, and `PeerQualifier` qualifies the peer at the specified place. A `PeerQualifier` is either an agent identifier or a badge predicate. Note that at most one agent qualifies in the case of a single agent identifier, while several agents may qualify if a single badge predicate is specified. In that case a randomly picked agent is chosen. To avoid infinite blocking, the parameter `TimeOut` can be used to specify a timeout interval. The operation blocks until the session is established or a timeout occurs, whatever happens first.

The parameters `PeerQualifier` and `PlaceId` of the operation `PassiveSetUp()` are optional. If neither of both parameters is specified, the caller expresses its willingness to establish a session with any agent residing at any place. By specifying `PlaceId` and / or `PeerQualifier` the calling agent may limit the group of potential peers. For example, a group may

be limited to all agents wearing the badge “Stuttgart University” and / or that are located at the caller’s place. As pointed out above, before a session is established both participants must agree explicitly. An agreement for session setup is achieved if both agents issue matching setup requests. Two setup requests, say R_A and R_B of agents A respectively B, match if

- `Place_Id` in R_A and R_B identifies the current place of B and A, respectively, and
- `PeerQualifier` in R_A and R_B qualifies B and A, respectively.

If a setup request issued by an agent matches more than one setup request, one request is chosen randomly and a session is established with the corresponding agent. A combination of `PassiveSetUp()` and `ActiveSetup()` allows a client / server style of communication (see Figure 5.4). The agent playing the server role once issues `PassiveSetUp()` when it is ready to receive requests. When an agent playing the client role invokes `ActiveSetup()`, this causes the `SetUp()` method of the server side to be invoked implicitly. `SetUp()` implicitly establishes a session with the caller and assigns a thread for handling this session. Therefore, once the server agent has called `PassiveSetup()`, any number of sessions can be established in parallel, where session establishment is purely client driven. If both agents issue (matching) `ActiveSetUp()` requests this corresponds to a rendezvous. Both requesters are blocked until the session is established or timeout occurs (see Figure 5.4b). This type of session establishment is suited for agents that want to establish peer-to-peer communication relationships with other agents. Communication between agents is peer-to-peer if both have their own “agenda” in terms of communication, i.e., both decide - depending on their individual goals - when they want to interact with whom in which way.



a) Client/Server-Interaction b) Peer to Peer Interaction

Figure 5.4 Different Types of Interactions

As pointed out above, Remote Method Invocation (RMI), the object-oriented equivalent to RPC, seems to be the most appropriate communication paradigm for a client / server style of interaction, while message passing is required to support peer-to-peer communication patterns. The available communication mechanisms are realized by so-called *Com* objects. Currently, there are two types of *Com* objects, RMI objects and messaging objects. *Com* objects are associated with sessions. Each session may have an RMI object, a messaging object, or both. Each session object offers a method for creating *Com* objects associated with this session. With the RMI object the methods exported by the session peer can be invoked. It can be compared with a proxy object known from distributed object-oriented systems. With the messaging object,

messages can be conveyed asynchronously between the participants of a session. Messages are sent by calling the `send()` method. For receiving messages the `receive()` and `subscribe()`-methods are provided. The `receive()`-method blocks until a message is received or timeout occurs, whatever happens first. If the `subscribe()`-method is invoked instead, the incoming messages are handed over by calling the `message()`-method of the recipient and passing the message as method parameter. The advantage of having the concept of *Com* objects is twofold. Firstly, only those communication mechanisms have to be initiated that are actually needed during a session. Secondly, other mechanisms, such as streams, can be added to the system transparently. The latter advantage enhances the extensibility of the system.

At any time, a session can be terminated unilaterally by each of the both session participants, either explicitly or implicitly. A session is terminated explicitly by calling `SessionObject.Terminate()` (see Figure 5.3), and implicitly when a session participant moves to another place. When a session is terminated, this is indicated by calling the `SessionTerminated()`-method exported by agents. Moreover, all resources associated with the terminated session are released. We want to mention, that for easier programming, we still allow the programmer to use “traditional” RMIs or messages without the need of a session overhead, giving them the opportunity to issue single communication acts.

5.4 Agent Infrastructure

5.4.1 Resource Manager

Resource management is necessary for two purposes. One is accounting, the other is resource control. Accounting is a prerequisite for commercial applications with agents, and resource control is necessary to prevent e.g. denial-of-service attacks. In Mole the following resources are managed:

- CPU time
- local network communication
- communication with remote networks
- number of created children
- total time at the local place

The CPU time used is calculated by counting the time slices given to threads of an agent. Mole has a central scheduler, the MCP (Master Control Process), that schedules all threads in the Mole system. We decided to implement our own scheduler, when problems with Java 1.0 led to the conclusion that the Java scheduler of the Solaris implementation had problems with concurrent execution of more than four threads of the same priority. This results from the Java specification being imprecise in this respect. The network communication is an important cost factor. Thus it is important for both accounting and resource control. Because all agent communication has to use the mechanisms provided by the agent system, control is done here. When an agent arrives at one place the arrival time is logged. This way the total time at the local place can be computed without problems. One other resource is not managed, the memory consumption of

an agent. While in principle extremely important, it can not be implemented with acceptable costs without modifying the Java virtual machine.

5.4.2 Directory Service

A directory service is an electronic database that contains information on entities. An example for a full-fledged directory service is X.500 (see e.g. CHADWICK (1994)). In our Mole system we simply provide a local directory service. It supplies information on agents providing a service denoted by a string. This local directory service exists on every place. An agent can register itself locally if it provides a service by submitting a string identifying the service to the directory service. Another agent wanting to use this service first asks the directory service. The directory service returns a list containing all agents providing the service. This list is either empty, or contains one or more agent identifiers. The agent now chooses one of the agent identifiers and contacts the agent.

5.4.3 Security Model

In Mole, the *Sandbox* security model is enforced by implementing a simple concept. In Section 2 we presented our agent model, and with it mobile and service agents. Mobile agents are the normal user agents, programmed and employed by the user. They have absolutely no access to the underlying system. Service agents are agents with access to system resources, providing controlled, secure abstractions of these resources inside the agent system. Furthermore, service agents may offer access to legacy software, using the native code interface offered by Java. This does not cause any security problems, because the service agents are immobile and may be started only by the administrator of the place. User agents may only communicate with other agents and have no direct access to system resources.

Additionally it can be decided on a per-place basis which types of agents to allow on a place. Only agents that are derived from a specific type can migrate to a place. This mechanism can be used to implement access restrictions. Take e.g. a place that allows only agents of a very specific type. These can only be created at one other, open place. Then every agent wanting to access a service on the first, closed place has to migrate to the open place and request a service. This service then creates one of the specific agents that migrate to the closed place.

5.5 Graphical Agent Monitor

In Mole the graphical agent monitor Moleview is used for the examining places, agents, and messages sent between them (see Figure 5.5). The snapshot has been made while running part of the test suite at installation time.Every Moleview window contains the information for one of the places that are inspected. In this example the place *location1.mole.de* has been examined. We can see 7 agents on the place, their ids and their descriptions. In the lower left corner of the window the services provided by agents are printed (in the example none). In the lower right corner of the window the communication in which the agents on this place participate is logged. This might be local or remote communication. RPCs as well as messages are logged with their message id, sender and receiver. Additional information about the agents, the provided services,

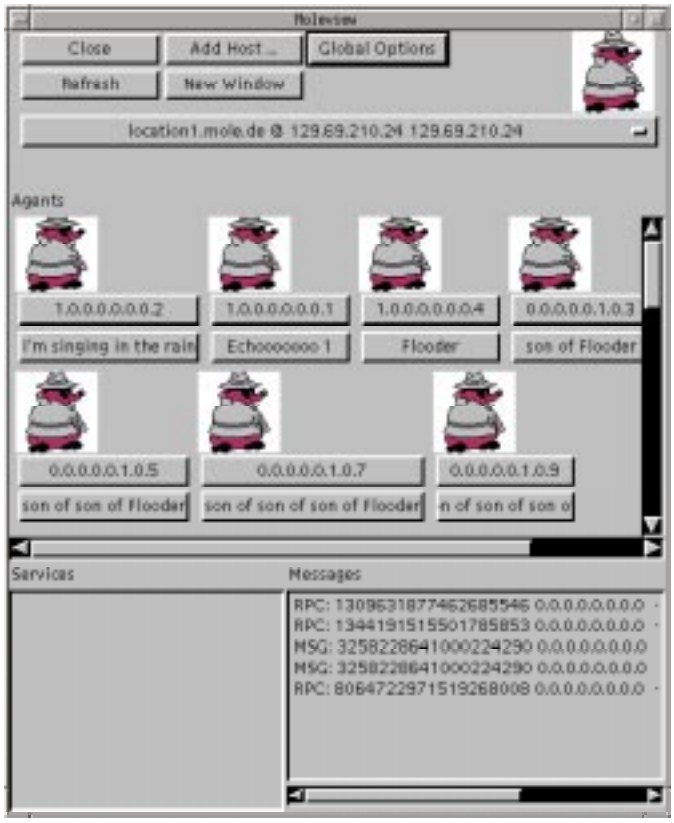


Figure 5.5 Moleview - The graphical Agent Monitor

the messages, and the RPCs can be acquired by using an inspector window. In this specific example a non-existing recipient is used for some of the tests for the communication subsystem.

5.6 Implementational Issues

5.6.1 Agent Identifiers and Name Resolution

In Mole, an agent is seen as a unique entity. This view is supported by using a globally unique name for every agent. This name is immutable, i.e. it does not change when the agent migrates. The uniqueness can only be guaranteed if the system creates the names used. If the system creates the agent identifiers, then these identifiers should be devisable without global knowledge. Additionally it is of advantage to be able to derive the site where the agent has been created from the agent identifier. Why do we place such constraints on the agent identifier? First, to be able to identify an agent (this is needed for communication, termination etc.), its name must be locally unique. Second, to be able to do the same after an agent has migrated, the name has to be immutable. From this follows that the agent identifier has to be globally unique. This can only be guaranteed if the system itself provides a service to create agent identifiers conforming to these requirements. If global knowledge is needed to create this agent name, then either an expensive mechanism has to be implemented to obtain the global knowledge, or a single point of failure is introduced if e.g. an identifier server is included into the system (see e.g. the Amoeba

sequencer poposed by TANENBAUM (1995)). The ability to derive the site where the agent has been created is of advantage e.g for localization algorithms utilizing the *home location registry* approach. This approach is used in GSM (*Global System for Mobile Communications*, see e.g. (MOULY and PAUTET, 1992)), where the identifier of the user (his telephone number) leads to a designated place (the home location registry) that contains the information how this user can be reached.

The agent identifier in Mole is created from information that can be obtained locally. Table 5.1 contains the components of the agent identifier. The internet protocol Version 6 address of the underlying system together with the port number of the engine allows to identify the engine on which the agent has been created. The uniqueness of the name is guaranteed by using a combination of a normal counter that is set to 0 at the start of the engine, and a so-called crash counter, that is incremented every time the engine is started. If more than 2^{32} agents are started the crash counter is incremented also. Two more bytes are reserved for future use, giving a total of 24 bytes.

Table 5.1 Components of the Agent Id

# Bytes	Meaning
4	Dynamic Counter, incremented for every new agent identifier
4	Crash counter, incremented every time the system is started. Also incremented if dynamic counter overflows.
12	IP Version 6 address of the system on which the engine runs
2	The port number of the engine
2	Reserved for future use (set to 0)

5.6.2 Thread Management Unit

One of the disadvantages of the Java language is that in some respects it is underspecified. One of the underspecified areas is the thread management. For instance if two threads of the same priority run in a virtual machine, nothing is said about the kind of scheduling used. It might even happen that one of the threads is executed solely and only when it has terminated is the other thread performed. This led to many problems in the platform-independent development of Mole, and we decided to implement our own thread scheduler. One of the few guarantees that Java gives is that a thread with higher priority is executed in preference to threads with lower priority. Our scheduler was implemented using this property of Java as follows: One designated thread called MCP is started when the Mole engine is booted, and controls all Mole threads (including all threads running in agents) at runtime. This thread gets the highest possible priority (apart from system threads). All other threads are on the lowest possible priority. The MCP-thread has a list of these threads and allows them their time slices by of changing their priorities. In regular intervals (time slices) the MCP-thread wakes up, lowers the priority of the running Mole thread, takes the next thread from its list and changes its priority to its own. Now it sleeps

for the length of the time slice, after which again the next thread is scheduled. This way the Mole system guarantees that threads are executed “concurrently” in time slices. A feature of the MCP is that the computing resources can be managed as well.

5.7 Using Java-Enabled Web Browsers to Run Mobile Agents

One of the main problems of the mobile agent technology is that before users get interested in using this technology, it has to offer advantages over the existing technology. One of the scenarios where mobile agents would offer tremendous advantages is their usage as an integrated service platform that can unify the existing information and service provider on the internet on the one hand and the need for automatic access to these things on the other side. In order to get such a platform, the technology, even a single product had to be wide-spreaded since no one would use such a platform without a large set of offered services, and no one would offer a service without a large set of users the service can reach. This need for a critical mass is one of the problems to solve in order to have this technology accepted. Another problem of current agent systems is that, as they rarely have been developed as a commercial product, the usage of such an agent system is rather complicated, at least for the end user. In order to allow more users to run mobile agent based application, the threshold to install an agent system has to be lowered.

For this purpose, an agent system based on Mole was developed that runs as a Java applet inside any Java-enabled browser. Although a Mole engine, the runtime environment of the Mole agent system, is a regular Java application, it cannot simply be executed by a browser. The reason for that lies in the area of security, namely the protection of computer from malicious applets. Since applets are currently not allowed in most browsers to do anything that can be used to attack the computer the applet is executed, typical restrictions for applets are:

- applets cannot access, start or manipulate other programs
- applets cannot open sockets except to the server they were loaded from
- applets cannot access system resources like file systems

Another problem is the different “operating mode” of web browsers and servers such as regular Mole engines: while the first were started and stopped by users at their will, servers normally are started once and run until either the computer crashes or a new version of a server is installed, both often in a controlled, non ad-hoc manner. A model that uses browsers to run an engine has to cope with the aspect that the user stops the browser at any time, regardless of the state of the agents. In this sense, browsers have similar characteristics like mobile devices.

The architecture of the modified engine has to reflect these problems, but it should also try to make these aspect transparent to the agents, so that mobile agents can run in both a browser and a server environment. To handle the restrictions of applets, a new component, called *Relay* was introduced (see Figure 5.6). The purpose of the Relay component is to act as a representative of the browser engine. Therefore, Relays are located on a regular Server Engine. At startup time, a Browser Engine opens a connection to a certain Relay. This connection, a socket, remains open until the Server Engine is stopped. Every time the Browser Engine wants to communicate with another engine or vice versea, the relay acts as a proxy object of the Browser Engine and relays the communicated data to or from the Browser Engine.

To do this in a transparent manner, Browser Engines use special location names of the form *browserLocation<engine number>_<location number>:<regular engine name>*, e.g. `browserLocation12_3:stuttgart.mole.informatik.uni-stuttgart.de`. These names are created dynamically by the Relay at Browser Engine startup time.Since the applet can open a connection to the computer from where the applet was loaded, a weeb server is also needed on the computer where the Realy resides. Fortunately, Mole engines already can be used as a web server for Mole classes since HTTP is used as the code transfer protocol between engines. The Browser Engine is configured by a list of parameters that are provided in the `<APPLET>` tag of the enclosing HTML page. The (not optimized) Browser Engine applet is as big as any regular Mole engine and needs some 40 seconds to load in a browser. Further optimization can reduce both the length and the loading time of the applet. To demonstrate the Mole Browser Engine technology, an existing game (Mister X) was configured using mainly the engine on the one browser an users uses to start the game.

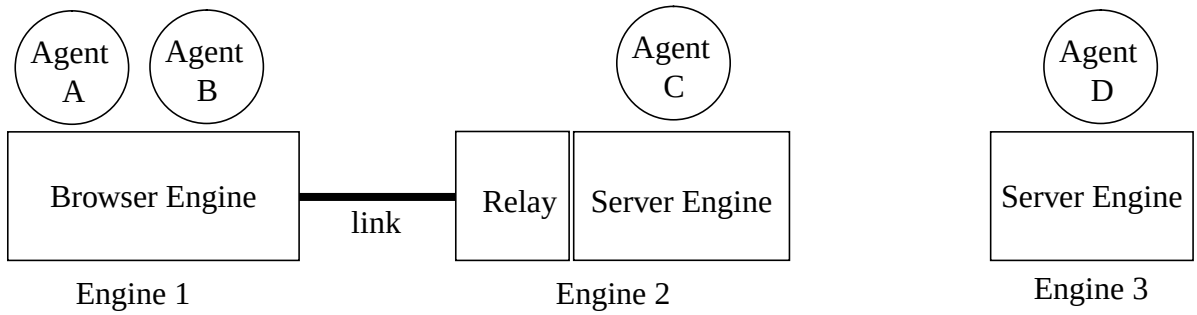


Figure 5.6 Architecture of the modified engine

5.8 Installation

Mole is a research development designed to experiment with new concepts arising in the context of mobile agents. It is available for free on the World Wide Web. The Java source code, documentation and sample agents can be downloaded from:

`http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html.`

This Web site currently allows downloading of Mole 2.1.2. Mole 3.0 is currently in the beta test phase and will be updated for the Middleware'98 conference in September 1998. For a prerelease of the beta version of Mole 3.0 please please contact Mr. Baumann, at University of Stuttgart, email: `Joachim.Baumann@informatik.uni-stuttgart.de.`

5.8.1 System Requirements

Mole is undemanding regarding hardware and software requirements, especially compared to Telescript (WHITE, 1997b). This is due to the design focusing on the use of an unmodified Java Virtual Machine and existing hardware. While this didn't allow e.g. strong migration or the control of the memory consumption of agents, it allows Mole to run on every hardware platform that runs a Java Virtual Machine version 1.1 or higher. We have tested the system on various

computer types and operating systems. Normal PC's with a Intel Pentium (we have not tested systems with Intel 486) or compatible CPU with main memory ranging from 16 to 128 MBytes under Windows 95, Windows NT V3.51 or V4.0, OS/2 or Linux runs the system as well as Sun Sparc with Solaris, IBM RS/6000 with AIX, or HP workstations with HPUX.

5.8.2 Configuration Files

Two main configuration files exist for Mole, the global mole resource file `globalmolerc`, and the user-specific Mole resource file "`~/.molerc`" (under Windows 95/NT "`C:\.molerc`"). These contain global as well as user specific definitions for variables that can be read by system agents at runtime. This way a simple method of configuring arbitrarily complex system agents is provided. Furthermore, many of the settings of the Mole system itself are defined in the Mole resource files. One example is the debug level, another one is the location of the global Mole installation. The following three variables are predefined:

- `$(HOME)` is the absolute path to the home directory of the user (under Windows 95/NT this is "`C:\`")
- `$(USER)` is the login name of the user
- `$(CWD)` is the working directory

A very handy feature for all of the Mole configuration files is that resources that have been defined already can be referenced in a simple way. Let us assume we have defined a resource "`Hello = howdy`" and another resource "`World = world`", then the following definition of the variable `HiWorld` would yield a contents of "`howdy world`": "`HiWorld = $(Hello) $(World)`". This is very advantageous for the definition of complex values. Moreover, a user can define user-specific values depending on global values set by the system administrator in the global resource file.

5.8.3 Starting a Sample Agent

In Appendix A a very simple agent that wanders between two places is listed (the static part is left out). The names of the places can be found in the two variables `home` and `target`. For the dynamic instantiation of agents a constructor without parameters is needed. As has been discussed in Section 3.1, the method `init()` is called first in the lifecycle of the agent. In this case we simply initialize the variables containing the two places and return the boolean value `true`, indicating that the initialization was successful. The system now calls the `prepare()`-method that sets the boolean variable `atHome` depending on the place of the agent (if at home, it is true). Now the agent is ready to start at the new place and its method `start()` is called. Here the agent simply decides where to migrate next. If the migration was not successful it prints an error message and dies. This agent can be brought into the system by issuing the following command at the command line of the Mole system (we assume two places, "`place1`" and "`place2`"):

```
new mole.apps.SimpleWanderer(Home place1, Target place2) at place1;
```

The agent is started as described above and begins to wander forth and back between the two places.

5.9 Conclusions and Future Work

We have presented a platform for Java-based mobile agents, a research development that implements many new concepts of mobile agent systems. Besides a basic infrastructure like weak migration of agents and local/global communication between agents and agent groups, the emphasis of the Mole system lies on providing a comfortable infrastructure for agents. In particular, Mole implements a resource manager for accounting and resource control, a directory service, a thread management unit and a global naming scheme for agents. The usage of Mole is supported by simple means for installation and configuration and by a graphical agent monitor.

Earlier Versions of Mole have been used e.g as the prototypical infrastructure for an electronic document system by KONSTANTAS, MORIN and VITEK (1996), as a simulation environment for distributed network management by Siemens in the project Swarms and as an execution environment for Tandem server classes by STRAßER *et al.* (1997). In order to make mobile agents usable in a commercial setting, our current research investigates further extensions to the system infrastructure. In particular, methods for orphan detection and agent termination have been investigated by BAUMANN (1997), coordination of agent groups has been investigated by BAUMANN and RADOUNIKLIS (1997), an approach for securing agents against malicious hosts has been investigated by HOHL (1997) and a protocol for preserving the exactly-once-property of mobile agents by ROTHERMEL and STRAßER (1997).

A Java-Code of the Wanderer Agent

```
package mole.apps;
import mole.*;

public class SimpleWanderer
    extends UserAgent
    implements MobileAgent
{
private LocationName home   = null;
private LocationName target = null;
private boolean atHome = null;

public SimpleWanderer()
{
}

public boolean init(Hashtable parameters)
{
    String des = (String)parameters.get("Description");
    if(des != null)
        super.init(des);// Set description of agent if given
    // in the real world all parameters would have to checked as above
    home = new LocationName((String)parameters.get("Home"));
    target = new LocationName((String)parameters.get("Target"));
    return true;
}
```

```
public void prepare()
{
   atHome = (getCurrentLocation().locationName()).equals(home);
}

public void start()
{
    if (atHome == true)
    {
        migrateTo(target);
        Engine.out("Wasn't able to migrate to " + target);
        die();
    }
    else
    {
        migrateTo(home);
        Engine.out("Wasn't able to migrate to " + home);
        die();
    }
}
}
```

References

BAUMANN, J. F. HOHL, K. ROTHERMEL, M. SCHWEHM, M.STRAßER 1998: Mole 3.0: A Middleware for Java-Based Mobile Software Agents, in Proc Int. Conf. Middleware '98 (to be published)

BAUMANN, J. 1997: A Protocol for Orphan Detection and Termination in Mobile Agent Systems. Bericht Nr. 1997/09 der Fakultät Informatik, University of Stuttgart, Germany.

BAUMANN, J. & RADOUNIKLIS, N. 1997: Agent Groups in Mobile Agent Systems. In Proceedings of the DAIS'97, Chapman & Hall, London, UK.

BRADSHAW, J. M. (Ed.) 1997: *Software Agents*. AAAI Press/MIT Press, Menlo Park, CA.

CHADWICK, D. (1994): *Understanding the X.500 Directory.* Chapman & Hall, London, UK.

COCKAYNE, W. R. & ZYDA, M. 1997: *Mobile Agents*, Manning Publ. Co., Greenwich.

FALCONE, J. R. 1987: A Programmable Interface Language for Heterogeneous Distributed Systems. *ACM Trans. Computer Systems*, 5(4):330-351.

GHEZZI, C. and VIGNA, G., Mobile Code Paradigms and Technologies: A Case Study. *In:* (ROTHERMEL and POPESCU-ZELETIN, 1997), pp. 39-49.

GRAY, R. S.1995: Agent Tcl: Alpha Release 1.1. Technical Report, Department of Computer Science, Dartmouth College.

GRAY, R. S. 1996: Agent Tcl: A flexible and secure mobile agent system. Proc. 4th Annual Tcl/Tk Workshop, pp. 9-23.

HOHL, F. 1997: An approach to solve the problem of malicious hosts. Bericht Nr. 1997/03 der Fakultät Informatik, University of Stuttgart, Germany.

HOHL, F.; KLAR, P. & BAUMANN, J. 1997: Efficient Code Migration for Modular Mobile Agents. *In:* Proc. 3rd ECOOP Workshop on Mobile Object Systems, dpunkt-Verlag, Heidelberg, Germany. (to appear)

IBM Aglets Workbench Team 1997:Aglets Workbench. *In:*(Cockayne & Zyda, 1997), pp.165-183.

JOHANSEN, D.; VAN RENESSE, R. & SCHNEIDER, F. B. 1994: Operating System support for mobile agents. *In:* Proc. 5th Workshop on Hot Topics in Operating Systems, IEEE Comp. Soc. Press, pp. 42-45.

JOHANSEN, D.; VAN RENESSE, R. & SCHNEIDER, F. B.1995: An Introduction to the TACOMA Distributed System. Technical Report 95-23, Department of Computer Science, University of Tromso, Finland.

KONSTANTAS, D.;MORIN, J. H. & VITEK, J. 1996: MEDIA: A Platform for the Commercialization of Electronic Documents. *In:* Tschiritzis, D. (ed.) 1996: *Object Applications.* University of Geneva, pp. 7-18.

MILOJICIC, D.; BREUGST, M.; BUSSE, I.; CAMPBELL, J.; COVACI, S.; FRIEDMAN, B.; KOSAKA, K.; LANGE, D.; ONO, K.; OSHIMA, M.; THAM, C.; VIRDHAGRISWARAN, S. & WHITE, J. (1998): MASIF: The OMG Mobile Agent System Interoperability Facility, (submitted to MA‘98).

MOULY, M. & PAUTET, M. 1992: *The GSM System for mobile Communication*. Europe Media Publications S. A., ETSI, Palaiseau, France.

MÜHLHÄUSER, M. (ed.) 1996: *Special Issues in Object-Oriented Programming*, dpunkt-Verlag, Heidelberg, Germany.

PEINE, H. & STOLPMANN, T. 1997: The Architecture of the Ara Platform for Mobile Agents. *In:* (ROTHERMEL & POPESCU-ZELETIN, 1997) pp. 50-61.

PERRET, S. & DUDA, A. 1996: Implementation of MAP: A system for mobile assistant programming. *In:* Proc. IEEE Int. Conf. of Parallel and Distributed Systems.

ROTHERMEL, K. & POPESCU-ZELETIN, R. (eds.) 1997: *First Int. Workshop on Mobile Agents*, MA’97, Lecture Notes in Computer Science 1219, Springer-Verlag, Berlin.

ROTHERMEL, K. & STRAßER, M. (1997): A Protocol for Preserving the Exactly-Once Property of Mobile Agents. Bericht Nr. 1997/18, Fakultät Informatik, University of Stuttgart, Germany.

RULIFSON, J. 1969: DEL. *In:* Internet Engineering Task Force, Network Working Group, Request for Comments 5, ftp://ds.internic.net/rdc/rfc5.txt

STRAßER, M.; BAUMANN, J. & HOHL, F. 1996: Mole: A Java Based Mobile Agent System. *In:* MÜHLHÄUSER, 1996), pp. 327 - 334.

STRAßER, M.; BAUMANN, J.; HOHL, F.; RADOUNIKLIS, N.; ROTHERMEL, K. & SCHWEHM, M. 1997: ATOMAS: A Transaction-oriented Open Multi Agent-System. Annual Report. Bericht Nr. 1997/14 der Fakultät Informatik, University of Stuttgart., Germany

TANENBAUM, A. 1995: *Distributed Operating Systems.* Prentice Hall, Englewood Cliffs, NJ, USA.

TSCHUDIN, C. F. 1993: On the Structuring of Computer Communications. Ph.D. Thesis, University of Geneva, Suisse.

WHITE, J. E. 1997a: Mobile Agents. *In:* (BRADSHAW, 1997) pp. 437-472.

WHITE, J. E. 1997b: Telescript. *In:* (COCKAYNE & ZYDA, 1997) pp. 37-57.

ZANDER, J. 1981: SOFTNET - Packet Radio in Sweden. In: ARRL Amateur Radio Computer Networking Conferences 1-4, The American Radio Relay League, Newington, CT, reprinted 1985, pp. 1.7 - 1.10.

6 WP 2.5: An Orphan Detection Protocol for Mobile Agents

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.

6.1 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: Section 6.2 presents our agent model. In Section 6.3 the shadow protocol is presented with different extensions and optimizations. Section 6.4 presents related work, and in Section 6.5 the conclusion and outlook is given.

6.2 The Agent Model

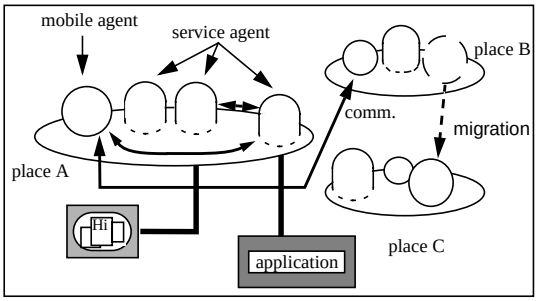


Figure 6.1 The Agent Model

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*.

6.3 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.

6.3.1 The Idea

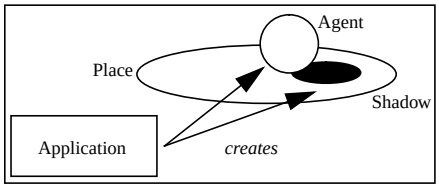


Figure 6.2 The Creation of a Shadow

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 (Figure 6.2). The agent is dependent of the shadow instead of the application. 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, the agent is declared to be an orphan and is removed.

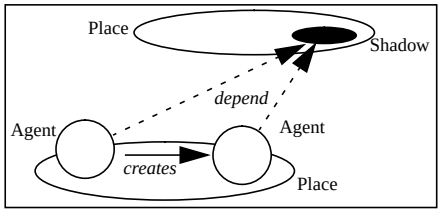


Figure 6.3 Creating a New Agent

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 (Figure 6.3). 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.

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 (Figure 6.4). 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 (Figure 6.5:). 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.

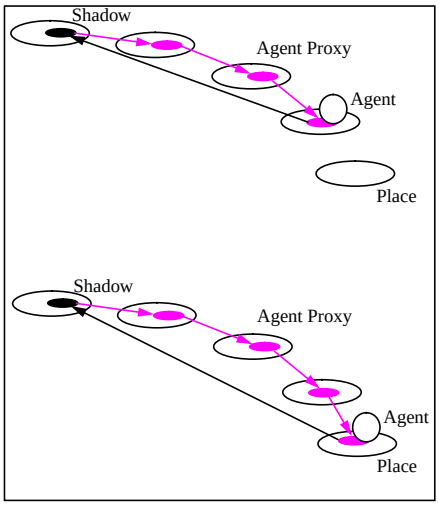


Figure 6.4 Proxie Paths

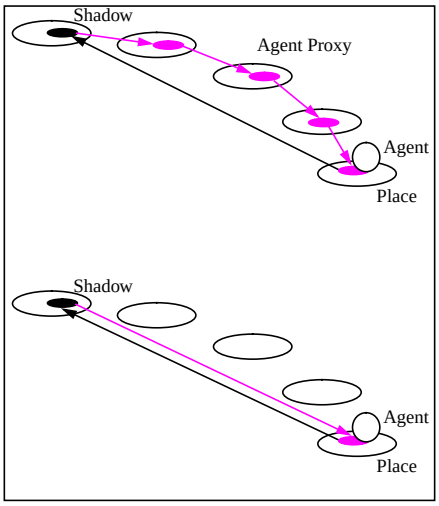


Figure 6.5: Regular Update of Proxies

6.3.2 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* (Figure 6.6). 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)
    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 6.6 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 Figure 6.7). 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 Figure 6.6). 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;
    proxyList.setTime(agentId, timeToLive);
```

Figure 6.7 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, ager
                                timeToLive);
        shadowproxy.add (agententry);
    return agententry.timeToLive;
remove(agentId)
[implement policy]
    agentproxy = listOfProxies.find(agentId);
    agentproxy.remove(agentId);
```

Figure 6.8 Methods in the Shadow Object

In Figure 6.8 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 Figure 6.9). 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. 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 Figure 6.9) is called to react to it.

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

Figure 6.9 System: Reaction to Timeouts

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 Figure 6.10).

```
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)
    if (error == notFoundError)
        return(error);
```

Figure 6.10 Finding Agents

6.3.3 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 Figure 6.11). 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);
    delete(this);
```

Figure 6.11 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 Figure 6.12). 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 is not received until the timeout, the shadow is removed (more complex reactions with retries can be chosen instead).

```
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,
                          shadow);
```

Figure 6.12 Extended System Methods: Regular Intervals

As soon as it is received, the timer is stopped and the *ttl* is set (see Figure 6.13). 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,
                      shadow.timeToLive);
```

Figure 6.13 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 Figure 6.14).

```
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(agentId
                  shadow, agentId);
            sendAllowance(from, place.name(),
                        agentId, timeToLive);
receiveAllowance(shadowPlace, agentId, timeToL
    stopTimer(agentId);
    agent = agentList.findAgent(agentId);
    agent.timeToLive = timeToLive;
    agent.shadowHome = shadowPlace;
    proxyList.setTime(agentId, timeToLive);
```

Figure 6.14 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 Figure 6.15).

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

Figure 6.15 Changed System Methods Detecting terminated agents

6.3.4 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 effec-

tively 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.

6.3.5 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 Section 6.3 , 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 2*ttl* 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.

6.4 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 ac-

count. 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].

6.5 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.

Comment: This paper does not contain the full protocol as an appendix due to space restrictions. For the complete description please refer to [3].

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.

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, K. Rothermel. “The Shadow Approach: An Orphan Detection Protocol for Mobile Agents“, Tech. Report 1998/08, Fac. of Computer Science, U. of Stuttgart, 1998.
4. 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.
5. 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.
6. 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.
7. General Magic, “Odyssey Web Site”. URL: http://www.genmagic.com/agents/
8. IBM. “The Aglets Workbench”. URL: http://www.trl.ibm.co.jp/aglets/
9. F. Mattern. “Verteilte Algorithmen”, Springer-Verlag, 1989.

10. G. Tel, F. Mattern. “The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes.“, ACM TOPLAS 15:1, pp. 1-35, 1993.
11. M. Shapiro, P. Dickman, D. Plainfossé. “SSP Chains: Robust, Distributed References supporting acyclic Garbage Collection”, Tech. Report No. 1799, INRIA, Rocquencourt, Frankreich, 1992.
12. M. Straßer, J. Baumann, F. Hohl. “Mole - A Java Based Mobile Agent System”, in Workshop Reader ECOOP ’96, d-punkt, pp. 327 - 334, 1996.
13. G. Tel. „Distributed Algorithms“, Cambridge University Press, 1994.
14. J. E. White. “Telescript Technology: The Foundation of the Electronic Marketplace”, General Magic, 1994.