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

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

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. Annual Report**

*M. Straßer, J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, M. Schwehm*

Bericht 1997/14
August 1997

# ATOMAS:

## A Transaction-oriented Open Multi Agent-System

## Annual Report

**A Project Funded By Tandem Inc., Cupertino**

16.7.1997

**Authors:**

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

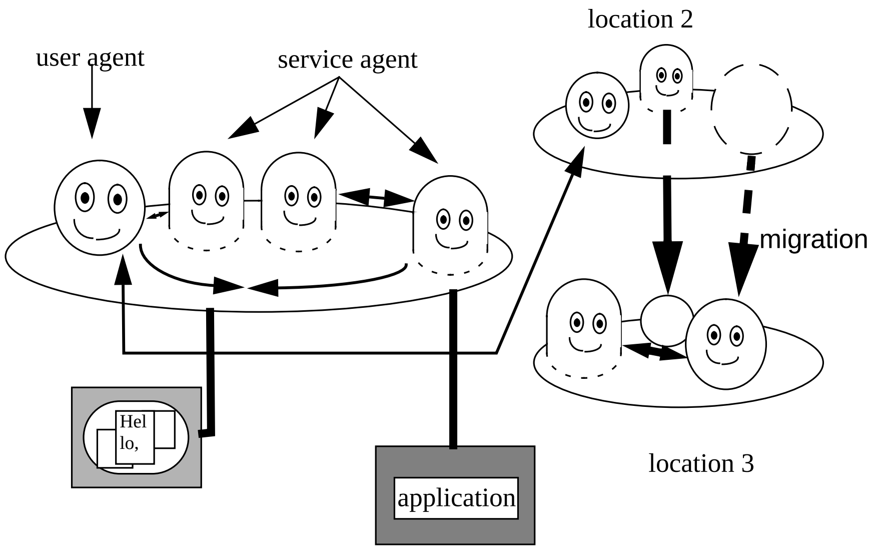


**Figure 1.1.** Mobile Agent System

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 first year. Section 2 provides an overview over the objectives of the work packages for the first year and their completion state. Sections 3-7 present the results in detail.

# 2 Workplan and Project State

In the first year, the necessary concepts for an open agent system architecture supporting remote execution had to be developed. These had to be based on the requirements of all partners. Therefore, four work packages had been identified (Section 2.1 - Section 2.4). Due to current standardization activities, WP 1.3 (see Section 2.3) was relocated to the second phase. Instead, work in two important areas, the efficient integration of services in agent systems (Section 2.5) and security (Section 2.6) was invested.

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

## 2.1 WP 1.1: Design and Implementation of Communication Subsystem

One of the basic capabilities of an agent is the communication to other agents and its environment. In this work package, the communication subsystem had to be designed and implemented exploiting existing base services. Hereby, the application of widely available protocols should be evaluated.

Based on the experience gained by a first implementation of a set of communication mechanisms, a study about “Communication Concepts for Mobile Agent Systems” has been written and published [BH+97]. Therein, different types of agent communication are classified and mechanisms for this communication types are identified. Beyond the existence of RPC-like mechanisms and message passing mechanisms, event mechanisms are identified as a prerequisite for agent group interaction.

This process (first implementation, evaluation delivering requirements) takes some time. Therefore, the work package hasn’t finished yet. Currently, existing base services and widely available protocols are analysed on their applicability to realize these communication mechanisms.

Section 3 contains a overworked version of the aforementioned study. We give an overview over the communication mechanisms already implemented in the agent system and other mechanisms we will implement in the near future

## 2.2 WP 1.2: Design and Implementation of Remote Execution Functionality

This work package aims at developing a representation of agent code that can be injected into a running system, and mechanisms to transfer and install code efficiently over a heterogeneous network.

Evaluating Java as implementation language for agents showed, that the remote execution functionality (i.e. the start of an agent on a remote system) and even a simple form of migration can be implemented using standard Java functionality. The prototype implementation uses the Java byte code as transport format for the agent code and the Java object serialization for the representation of the agent data.

The performance of the migration/remote execution functionality depends heavily on the time for the transport of the code to the remote system. Therefore, efforts were invested in the research on efficient code migration in an open agent system. The results have been published at the ECOOP’97 Workshop on Mobile Object Systems.

Section 4 contains a description of the mobility mechanisms and the paper on efficient code migration.

## 2.3 WP 1.3: Integration of OLTP into Implemented Remote Execution Environment

A precondition for the development of reliable business applications is the ability to use transactions. In this work package, services to initiate and terminate transactions and to access database services should be integrated. Furthermore, a service to execute code/agents in server classes should be implemented.

In the Java developer community, there are currently lots of activities in the environment of OLTP. With the release of JDK 1.1, JDBC [Sun96b][Sun96c], a standard SQL database access interface, providing uniform access to a wide range of relational databases, is now available. JTS [Sun96d], the Java Transaction Service API is currently developed by SUN (in collaboration with IBM, Tandem and BEA) and is currently available as version 0.5. It is a low level interface, which is compatible with the Object Management Group's Transaction Service (OTS) specification. Enhancements to OTS include, among others, the specification of transaction context propagation through communication protocols.

In order to use standard facilities, the integration of these OLTP services was relocated in the second phase of the project. The execution of code/agents in server classes will be realized as soon as Java for Tandem OSS is available.

## 2.4 WP 1.4: Performance Model and Verification

Increasing efficiency by moving the computation to the data (and therefore by reducing communication via slow networks) is one of the common arguments given for the use of mobile agents. In this work package, a performance model had to be developed which helps to decide when to use remote execution versus remote procedure call.

Section 5 contains the paper about the performance model published at the International Conference on Parallel and Distributed Processing Techniques and Applications 1997 (PDPTA'97)[SS97].

## 2.5 Efficient Integration of Services

The electronic marketplace of the future will consist of a large number of services located on open, distributed and heterogeneous platforms, 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 service by client agents.

In this electronic marketplace, it is important to efficiently provide services to be able to satisfy the requests of a large number of customers. In Section 6, we address the efficient integration of existing services and the efficient implementation of new services into Mobile Agent Systems.

## 2.6 WP 2.2 Requirement Analysis Concerning Security

In the area of electronic services, 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, but the anonymity of a worldwide communication network and the ease of the automatic exploitation of security gaps in electronic applications make it necessary to meet this demand when it comes to commercial transactions done by computers.

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

# 3 Communication System

Mobile agents are often described as a promising technology, moving towards the vision of usable distributed systems in widely distributed heterogeneous open networks. Particularly, its promise to offer an appropriate framework for a unified and scalable electronic market has led in the past years to a great deal of attention. Since the deployment of mobile agent systems in a large scale is crucial for the success of this technology, the emerging problems and needs have to be well understood.

A fundamental question tightly related to communication is how mobile agents are identified. On the one hand, there is certainly a need for globally unique agentIds. Identifier schemes that provide for migration transparency are well-understood today. However, such a scheme might be too inflexible in agent-based systems. Assume for example, that a group of agents cooperatively perform a user-defined task. Assume further that one group member wants to meet another member of this group at a particular place for the purpose of cooperation. In this case, the member should be identified by a (placeId, groupId) pair. If the agent to be met additionally is expected to play a particular role in this group, the identifier would have the form (placeId, groupId, roleId). For supporting those application-specific naming schemes we propose the concept of badges.

For the purpose of cooperation mobile agents must ’meet’ and establish communication relationships from time to time. For this purpose, we propose the concept of a session, which is an extension of Telescript’s meeting metaphor. A number of the currently existing agent systems are purely based on an RPC-style communication. While this type of communication is mostly appropriate for interactions with service agents, i.e. those agents that represent services in the agents’ world, it has its limitations if agents interact like peers. Therefore, we propose to support both message passing and remote method invocations.

In the general case, a group of agents performing a common task may be arbitrarily structured and highly dynamic. In those environments, one can not assume that an agent that wants to synchronize on an event (e.g., some subtask this agent depends upon is finished) knows a prior which agent or agent subgroup is responsible for generating this event. Therefore, we suggest to use the concept of anonymous communication, allowing agents to generate events and register for the events they are interested in, as a foundation for agent synchronization.

## 3.1 Types of Agent Communication

In this section, we will address the various types of communication. Considering inter-agent interaction, we have to distinguish between following types of communication:

1. Agent/service agent interaction
   Since service agents are the representatives of services in the agent world, the style of interaction is typically client/server. Consequently, services are requested by issuing requests, results are reported by responses. To simplify the development of agent software, an RPC-like communication mechanism should be provided.

2. Mobile Agent/Mobile Agent Interaction
   This type of interaction significantly differs from the previous one. The role of the communication partners are peer-to-peer rather than client/server. Each mobile agent has its own agenda and hence initiates and controls its interactions according to its needs and goals. Furthermore, the communication patterns that may occur in this type of interaction might not be limited to request/response only. For example, assume a mobile agent passes a form

to another agent and then terminates. The receiving agent would fill out that form by using various services and finally would deliver the filled out form to another agent waiting at some previously specified place. The required degree of flexibility for those interactions is provided by a message passing scheme. Even higher-layer cooperation protocols, such as KQML/KIF [FMM94], are based on message passing.

3. Anonymous agent group interaction
   In the previous two types, we have assumed that the communication partners know each other, i.e. the sender of a message or RPC is able to identify the recipient(s). However, there are situations, where a sender does not know the identities of the agents that are interested in the sent message. Assume, for example, a given task is performed by a group of agents, each agent taking over a subtask. In order to perform their subtasks, agents itself may dynamically create subgroups of agents. In other words, the member set of the agent group responsible for performing the original task is highly dynamic. Of course, the same holds for each of the subgroups involved in this task. Now assume that some agent wants to terminate the entire group or some subgroup. In general, the agent that has to send out the terminate request does not know the individual members of the group to be terminated. Therefore, communication has to be anonymous, i.e., the sender does not identify the recipients. This type of communication is supported by group communication protocols (e.g., see [BvR94]), the concept of tupel spaces [CG89], as well as sophisticated event managers. In the latter approach, senders send out event messages anonymously, and receivers explicitly register for those events they are interested in.
4. User/Agent Interaction
   Although a very interesting area of research, the interaction between human users and software agents is beyond the scope of this text. For a discussion of this type of communication the reader is referred to e.g., [Mae94].

Let us briefly summarize our findings. Different types of communication schemes are needed in agent-based systems. Besides anonymous communication for group interactions, message passing and an RPC-style of communication is suggested. In our model, message passing and RPC is session-oriented, which means that agents that want to communicate have to establish a session before they can send and receive data. In the remainder of this section, we investigate event managers for anonymous communication and discuss the concept of session-oriented communication in the context of agent-based systems.

## 3.2 Session-Oriented Communication

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 agent_Ids introduced in Sec. 1 and so-called badges.

Agent_Ids are well-suited for identifying service agents. We assume that there exists a directory system, that maps user-defined service names to service agent_Ids. 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 our system.

In the case of mobile agents the concept of agent_Ids 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_Ids were available, both agents would have to know each others ids. Actually, for identification it would be sufficient to say „At place XYZ I would like 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“. 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_Id, badge predicate*)-pair, which identifies all agents fulfilling the *badge predicate* at the place identified by *place_Id*). 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).

Now let us take a closer look to 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. Limiting sessions to intra-place relationships seems to be too restrictive. There are many situations, where it is more efficient to communicate from place to place (i.e., generally over the network) than migrating the caller to the place where the callee lives. Consequently, we feel that the mobility of agents cannot replace the remote communication in all cases.
- In order to preserve the autonomy of agents, each session peer must explicitly agree to participate in the session. Further, an agent may unilateraly 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 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 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 intend 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.

## Session Establishment

In order to set up sessions two operations are offered, *PassiveSetUp* and *ActiveSetUp*. (see Fig. 3.1). 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.

```
PassiveSetUp({PeerQualifier}, {PlaceId})-> nil
ActiveSetUp(PeerQualifier, PlaceId, Timeout) -> SessionObject
Terminate(SessionObject) -> nil
SetUp(SessionObject)
```

**Figure 3.1** Session Methods

In the case *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_Id or a badge predicate. Note that at most one agent qualifies in the case of a single agent_Id, while several agents may qualify if a single badge predicate is specified. To avoid infinite blocking, 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.

Parameters *PeerQualifier* and *PlaceId* of 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 specifiying PlaceId and/or PeerQualifier the calling agent may limit the group of potential peers. For example, this 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 explictly. 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 resp. B, match if

- *PlaceId* in $R_A$ and $R_B$ identifies the current location 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 Fig. 3.2). 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.

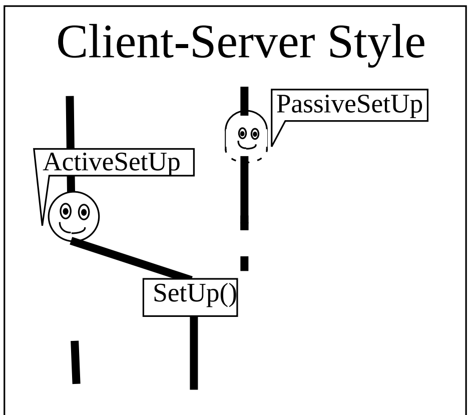


**Figure 3.2** C-S interaction

If both agents issue (matching) ActiveSetUp requests this corresponds to a rendezvous, both requestors are blocked until the session is established or timeout occurs (see Fig. 3.3). 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.

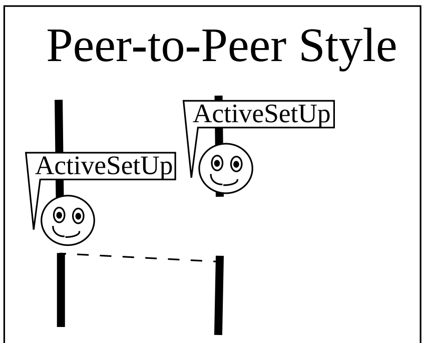


**Figure 3.3** P-P interaction

### Communication

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. Figure 3.4 shows the RMI object enabling access to methods alpha and beta of object B.

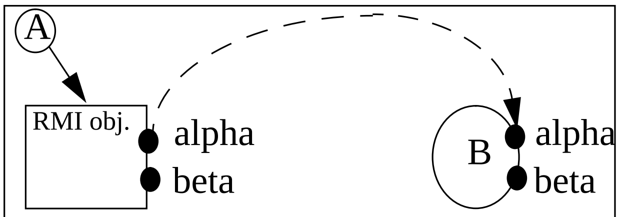


**Figure 3.4** RMI obj.

With the ***Messaging object***, messages can be conveyed asynchronously between the participants of a session (see Fig. 3.5). 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.

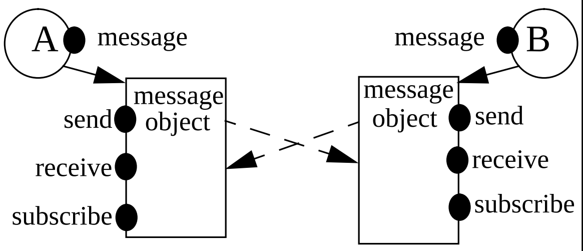


**Figure 3.5** message obj.

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, and secondly, other mechanisms, such as streams, can be added to the system. The latter advantage enhances the extensibility of the system.

### Session Termination

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 *Terminate* (see Fig. 3.1), 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 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.

After we saw a session-oriented communication scheme for one-to-one agent interaction, we will now investigate an anonymous communication scheme that is used for group interactions.

## 3.3 Anonymous Communication at the example of agent synchronization

Two widely deployed concepts for anonymous communication are tuple spaces and sophisticated event managers. In contrast to the blackboard concept, tuple spaces provide additional access control mechanisms. Agents employ tuple spaces to leave messages without having any knowledge who will actually read them. For a discussion of the tuple space concept the reader is referred to [CG89] or [LDD95]. In the remainder of this chapter we will concentrate on event mechanisms as a well-suited concept for inter-agent synchronization.

Applications can be modelled as a sequence of reactions to events, that in turn generate new events. Events may be user- (e.g. reaction to a message), application-, or system-initiated (e.g. signal sent by a process). An event-based view maps quite closely onto real life, and any programming primitives that support event-based concepts tend to be more flexible in modelling a given problem.

The event model is particularly well-suited for distributed communication since it abstracts from the receiver’s identity. As a consequence, it enables the specification of complex interactions without the need to know the communication partners in advance. With regard to agent systems, the event model simplifies application- as well as system-level communication. On the application level, events are employed as a general communication means. On the system level, events can be used to design and implement protocols that encompass agent synchronization, termination, and orphan detection.

In this chapter, we will examine the suitability of event management as an infrastructural component for inter-agent synchronization. Therefore, we will first define our notion of events. Based on this notion the concept of synchronization objects is presented and an application scenario explored, where synchronization is managed through this technique. Finally, a brief overview of the OMG event model is given and the concept of synchronization objects described by employing the OMG terminology.

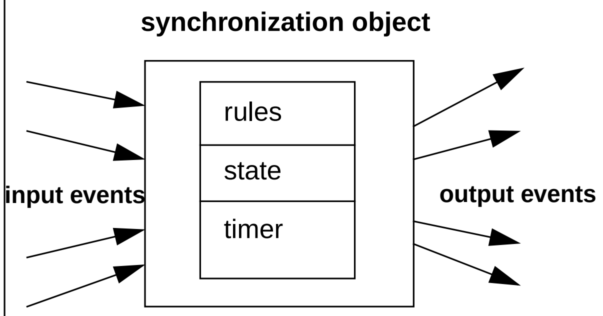
### 3.3.1 Events

In our notion, events are objects of a specific type, containing some information. Events are generated by so-called producers and are transferred to the consumer by the event service. Consumers (and, depending on the concrete implementation of the event service, also producers) have to register at the event service for the type of events they want to receive or send.

As consumers and producers may only interact if both know which events to produce or to consume, they necessarily have to share common knowledge of the used event types in an interaction group. For this, there exist two alternatives: Either the event types are negotiated at startup time, then this information configures the agents before a migration, or the event types have to be communicated to the members of the interaction group.

### 3.3.2 Synchronization objects

Synchronization objects (Fig. 3.6) are defined as active components responsible for the synchronization of an entire application or only parts of it. Synchronization objects monitor specific input events. Depending on these events, internal rules, state information and timeout intervals, output events are generated, that in turn may be the input for other synchronization objects.



```
Rule 12: if ((event98 and event51) or
           (event3 and variable1==true)) then
         {
         send(event90);
         variable1=false;
           }
variable1: boolean
timer1 = after 10 seconds { send(event51) }
```

**Figure 3.6** sync. object

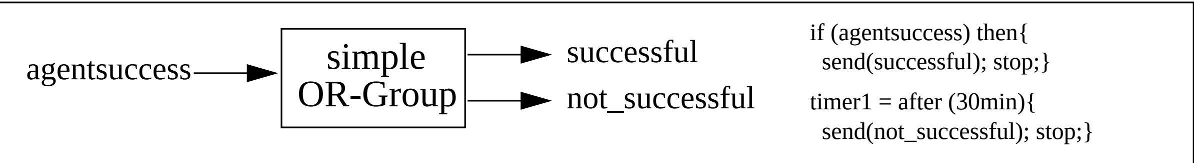
Rules are arbitrarily complex expressions triggered through input events. They consist of a condition and an action part. The condition part is a logical expression composed of event types and state information of the synchronization object. If the logical condition becomes true, the action part is triggered. The action part itself consists of simple commands (e.g. send output events, change internal state, stop the synchronization object to process events). The state consists of a set of variables. Timers are special rules with no input events that trigger actions after a specified amount of time.

An agent group comprises logically related agents. Syncronization objects are well-suited to model dependencies within agent groups. Relationships between agents are expressed by the synchronization object's internal rules and can be defined in terms of success (i.e. a group is only successful if a well defined set of the group members have succeeded). Agents participating in such groups send success events after they have accomplished their task. The synchronization object receives success events and processes this input through its internal rules. As a result, output events are generated. In case a generated event is an success event it can be used to nest groups (i.e. an output event of one group is used as an input event of another group).

#### Example: OR and AND groups

Two agent group types of particular interest are the OR-group and the AND-group. For the OR-group's success it is sufficient if at least one group agent accomplishes its task. OR-groups are eligible for parallel searching in a set of information sources. As soon as one agent has found the required information the group has succeeded in its task.

A simple OR group (Fig. 3.7) includes only three event types. The input event *agentsuccess*, sig-

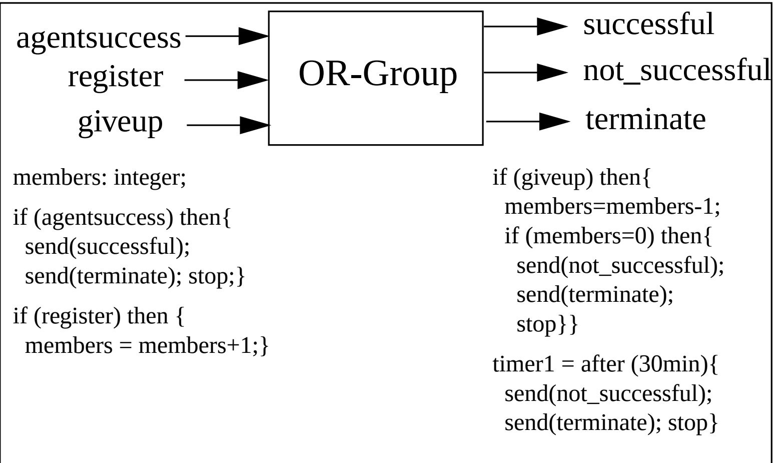


```
if (agentsuccess) then{
  send(successful); stop;}
timer1 = after (30min){
  send(not_successful); stop;}
```

**Figure 3.7** simple OR group

naling the success of an agent, and the output events *successful* and *not_successful*, signaling the group's success. The OR group employs only one rule and one timer. The rule causes the synchronization object to send an event signaling the group's success (*successful*) and to disable itself afterwards. If the timer fires first (e.g. caused by application specific timeouts or processing failures like deadlocks or crashes), the synchronization object signals *not_successful* and stops the processing.

The presented model is not very efficient: if one group member succeeds, all other group members are obsolete and, if all group members detect that they are not able to complete their task, the group fails. The definition of the OR-group illustrated by Fig. 3.8 takes these cases into account. Agents detecting that they cannot succeed, generate the *giveup* event. If all group members signal a *giveup*, the group fails. For this, the group has to know its members - either by keeping them in mind at the group's creation time or by registering group agents through the *register* event. In the latter case the number of members potentially being able to succeed is counted and stored in the state variable *members* (more sophisticated approaches could maintain agentId list, transmitted via the events and ensuring that only events from subscribed agents are accepted). If *members* becomes zero, the event *not_successful* is instantly generated. The *terminate* event (to terminate the group members) is generated if the group either succeeds or fails.



```
members: integer;
if (agentsuccess) then{
  send(successful);
  send(terminate); stop;}
if (register) then {
  members = members+1;}
```

```
if (giveup) then{
  members=members-1;
  if (members=0) then{
    send(not_successful);
    send(terminate);
    stop}}
timer1 = after (30min){
  send(not_successful);
  send(terminate); stop}
```

**Figure 3.8** OR group

Alternatively, crash events are used instead of timeout intervals. Crash events are reliable signals that are sent if agents are prevented to terminate successfully their processing (e.g. caused by network partitions, node crashes, or byzantine agent errors). Consequently, since agents can generate success or crash events, no timeout mechanism is further needed. However, crash event management is very hard to accomplish. The necessary surveillance protocols are very complex (see [Wa82], [HS80]) and do not consider migrating elements. Furthermore, mobile devices are hard to surveil due to their sporadic connection to the rest of the network.

In contrast to OR-groups, AND-groups succeed only if all agents have accomplished their task. AND-groups suit well for various scenarios (e.g. a customer wants to buy a flight, book a hotel and rent a car. For each subtask an agent is created and added to the AND-group. Only the success of all three subtasks together leads to a success of the AND-group). The structure of AND groups is very similar to the structure of OR-groups and therefore omitted here.

### 3.3.3 The OMG event model

The Object Management Group event services specification ([OMG94]) defines the Event Service in terms of suppliers and consumers. Suppliers are objects that produce event data and provide them via the event service, consumers process the event data provided by the event service. If a consumer is interested in receiving specific events, it has to register for them. This means a supplier of events knows who the recipients are (this does not exactly conform to the original definition of event mechanisms). Two communication models are supported between suppliers and consumers, the *push* model and the *pull* model. In both models all communication is synchronous. In the push model, a supplier pushes event data to the consumer, sending to each of the registered objects the event. In the pull model, consumers pull event data by requesting it from the supplier.

What makes this event service flexible and powerful, is the notion of the event channel. To a supplier, an event channel looks like a consumer. To a consumer on the other hand, the event channel seems to be a supplier. Furthermore, the communication model between the different participants can be chosen freely. By using an event channel, suppliers and consumers are decoupled and can communicate without knowing each other's identity. Suppliers and consumers communicate synchronously with the event channel but the semantics of the delivery are up to the designer of the specific event channel. Two types of channels are defined, typed and untyped channels. How these event channels are implemented is not defined in the OMG specification. By not imposing any restrictions on the semantics, the specification allows implementations to provide additional functionality in the event channel implementation. Persistent events (events that are logged) or reliable event delivery mechanisms come to mind. Because the event channel interface complies to the definition of the consumer's interface and to the definition of the supplier's interface, they can be chained without problems. This allows to build arbitrarily complex event channel hierarchies with a broad functionality.

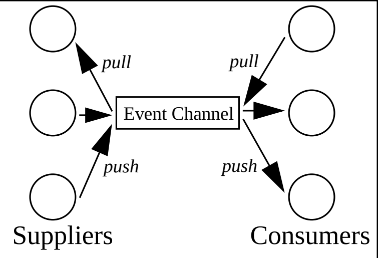


**Figure 3.9** Event Channels

Products following the OMG specification are commercially available (e.g. Iona OrbixTalk[ION96], or Sunsofts NEO [Sun96a]) or under development (IBM OpenBlueprint [IBM95]).

### 3.3.4 Synchronization using the OMG model

This section tries to map the presented group model onto OMG event services. Hereby, it is assumed that, in contrast to current implementations, event services support mobile participants. Support of mobile participants will be subject of future work.

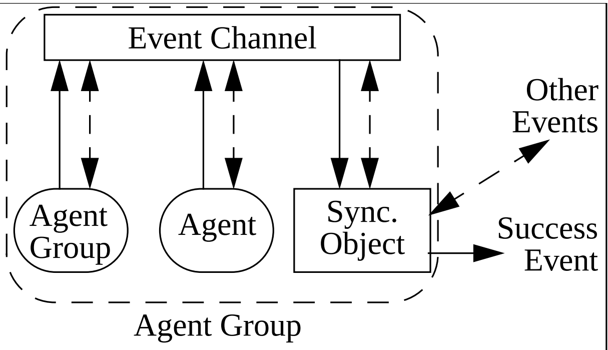


**Figure 3.10**

With the employment of an untyped event channel for group communication, OR and AND-groups can be implemented. The channel is untyped because different event types are transmitted through it. As the information about success is of foremost importance to the synchronization object, agents and synchronization object implement the push model. The synchronization object contains a reference to the event channel. The agent that creates the group has access to its synchronization object and thus the ability to forward the event channel reference to other agents, e.g. at creation time. The group members subscribe to the event channel as suppliers (e.g. for *agentsuccess* event) as well as consumers (e.g. for *termination* event). The communication to non group entities is handled by the synchronization object, either by sending the events directly to an agent (e.g. the parent agent creating the group) or by using another event channel (e.g. an event channel of a higher-level group).

## 3.4 Related Work

Current systems for mobile agents employ many communication mechanisms such as messages, local and remote procedure calls or sockets, but, at our knowledge, no system uses a global event management for communication and synchronization. There are “events” in AgentTcl [GCK96], but they are simply (local) messages plus a numerical tag.

Although the use of sessions offers certain advantages as shown above, existing agent systems barely provide session support. Telescript [GM96], for example, which introduced a kind of sessions by using the term meeting for mobile agent processing, offers only local meetings, that allow the agents only to exchange local agent references. The meet command is asymmetric, i.e. there is an active meeting requester, the “petitioner” and a passive meeting accepter, the “petitionee”. The petitionee can accept or reject a meeting, but only the petitioner gets a reference to the petitionee. Agents communicate after opening a meeting by calling procedures of each other (i.e. the petitioner can call procedures of the petitionee). As there is no possibility during the execution of a procedure to obtain information about an enclosing meeting, agents cannot access session context data. Furthermore, an agent can open only one meeting per agent as a petitioner. Finally, agents may migrate during meetings, and if an agent takes shared objects with it, the other agent will not see this until it tries to access a shared object and gets a “Reference void” execption. To summarize the Telescript meeting, we can say, that it is not a session according to our definition.

There are also “meetings” in ARA[Pei96] and in AgentTcl. Meetings in ARA build up communication relations between two agents over which (string) messages can be exchanged, meetings are local and the only supported “specification method” is anonymous addressing via meeting names. Meetings in AgentTcl are just a mechanism that opens a socket between two agents.

## 3.5 Communication Mechanisms in MOLE

The existing communication mechanisms in MOLE are messages and remote procedure calls. Both can be used not only in local communication, but in global communication as well.

Messages are sent with at-most-once semantics. The receiver is an agent at a specified location. Messages can be sent synchronously or asynchronously. If a message is sent asynchronously, the agent can continue at once. If the message is sent synchronously, the agent execution blocks until the message is delivered. The delivery of the message is done by calling a method called receiveMessage(Message) in the receiving agent. This method processes the incoming message.

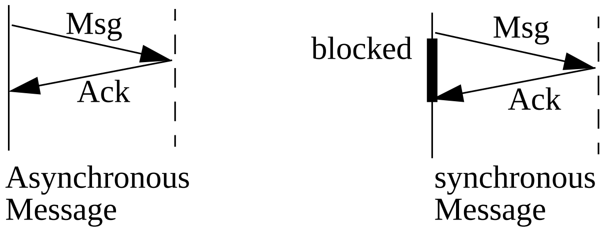


**Figure 3.11** Messages in Mole

We build on this message subsystem to realize RPC's. RPC's are implemented by sending a synchronous message to the target location, that in turn sends this message to a specific method in the agent. This method implementing the RPC processes the message, acts according to the request, and returns a result. This result is put into a message that is sent back to the originating location. The location unpacks the result and returns it to the agent, that continues its work. The delivery to the server agent is done by calling the method dispatch(RPCMessage) in the server agent. This method processes the RPC and returns the result.

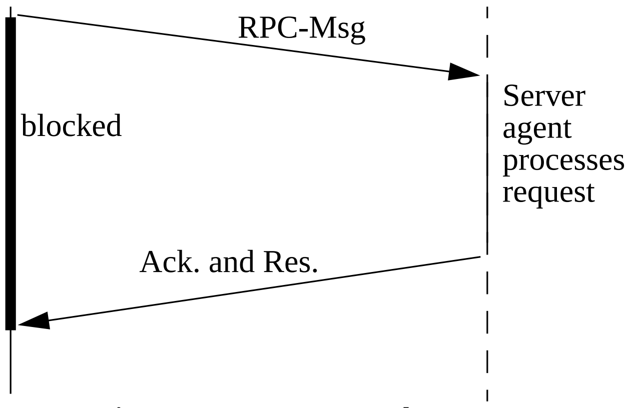


**Figure 3.12** RPC in Mole

We tried to use the standard Java RMI package (Remote Method Invocation), but in the version 1.02 of the Java JDK this is not correctly implemented. Thus we were forced to implement the RPC in the aforementioned way. As soon as the system will be adapted to the version 1.1 of Java, the RMI package will be tested again, and if found implemented correctly, will be used for the RPC.

### 3.5.1 Error Semantics

For delivering the messages (normal and RPC messages alike), the following error semantics are defined:

| | |
|---|---|
| • 0 | do nothing if the message cannot be delivered. |
| • 1 | send an error message (with error semantics 0) back to the sender if an error occured. |
| • 100 + x | try to deliver the message for x seconds after receiving it. If it cannot be delivered in that time, do nothing. |

### 3.5.2 Representation of Data

For transferring the data contained in the messages, we have to transform it to a machine-independent format. We decided to chose the standard for Java 1.1, which has been provided in beta version for Java 1.02, the serialization package. This package allows to create a machine-independent representation of any given object. Thus we can send all objects used in Mole within a message. In fact, the agent migration itself builds on the message subsystem.

### 3.5.3 Message API

#### The Message Object

```
public class Message
extends Object {
  public AgentName sender;  // the sender of the message
  public LocationName senderlocation;   // the location of the sender
  public AgentName receiver;  // the receiver of the message
  public LocationName receiverlocation; // the location of the receiver
  public int errorsemantics = 0;    // the error semantics of the message
                                                    // 0 = do nothing if this message can’t delivered
                                                    // 1 = send an error message (with errorsemantics 0) back
                                                    // to the sender if an error occured
                                                    // 100 + x  = try to deliver this message x seconds after
sending
                                               // even if the receiver or the receiverlocation isn’t available yet
  public long messageid = (new Random()).nextLong();    // the messageid of this message
  public Object content = ““;   // the content of the message
// ---------------------------------------------
public Message() {
} }
```

As can be seen the message object contains information about sender, senderlocation, receiver, receiverlocation, about the error semantics, a message id that is randomly generated for each message, and an object representing the contents of the message.

The RPCMessage contains an additional field for the name of the service requested in the RPC call, and a method to create a result message containing the result of the request.

#### The Methods

- `void getCurrentLocation().message(Message)`

  This method sends the given message to its destination. The method takes the message, starts a new message thread, and returns. The newly started message thread sends a sync-message to the destination.

- `void getCurrentLocation().syncmessage(Message)`

  This method sends the given message to its destination. While the message is sent, the sending agent thread is blocked.

### 3.5.4 RPC API

- `public Object call(RPCMessage rpcm)`

  This method sends the given RPCMessage to the server agent on the location specified in the RPCMessage. The server agent processes the RPC in its method dispatch(RPCMessage) and returns a result, that is delivered to the calling agent.

## 3.6 Future Work

Driven by the question how to identify potential communication partners and the need for well-suited communication schemes with regard to different types of agent interaction, we discussed two communication concepts in the context of Mobile Agent systems: sessions and the use of a global event management for infrastructural purposes.

Sessions establish either actively or passively a context for interactions. The communication partners are addressed either by globally unique agent identifiers or via badges. Agents can build several sessions simultaneously - even with the same communication partner. Communication in sessions is based on RPC or message mechanisms.

To bypass the problems arising from the need to communicate to potentially unknown group members performing the same task, we proposed the use of a global event management. The employment of events for the realization of a general synchronisation was shown. Therefore, we introduced the notion of synchronization objects, active components that offer different synchronization services. Using timers and state information, synchronization objects consumed, processed and produced events as input for other synchronization objects or other components. After a short overview of the OMG event model, the presented group model is mapped onto the OMG event services.

Existing implementations of event services already provide persistency (NEO and IONA OrbixTalk). But none of the existing implementations can cope with mobile participants. In order to support particular requirements imposed by mobile agents, appropriate event channel designs are required. While distributed event services with stationary participants are well understood, additional questions are raised by the mobility issue. The further exploration of this promising research field comprises the design and implementation of such distributed event services that support apart from different channel semantics also mobility of participants.

We are in the process of implementing a distributed event service for mobile participants and the session concept. While a first beta version of an event service already exists and is used in-house, the implementation of the session concept will need another 3 months.

# 4 Design and Implementation of Remote Execution Functionality

The aim of this workpackage was to design and to implement an efficient method to generate, transfer and install code across a heterogeneous network.

Since Remote Execution needs a possibility to parametrize the execution of code and since parameters have to be of an arbitrary type, we decided to also design and implement one form of agent migration, the so-called *Weak Migration* in this workpackage (which was originally scheduled for the second year). Weak Migration is the migration of the code and data state of an agent, i.e. the transport also of the (arbitrary) instance variables of the agent code, but not the transport of the execution state. In this Section, we will not distinguish between agents and remote execution units at the technical level, the term agent will always refer to the transported object.

## 4.1 Code representation

Due to the selection of Java as the agent language, the code representation (Java Bytecode) was given by the Java specification. Therefore, the generation and installation of code did not need to be designed by ourselves, but was offered efficiently by the Java Virtual Machine. The bytecode format is described in [LY97].

## 4.2 Generation of code

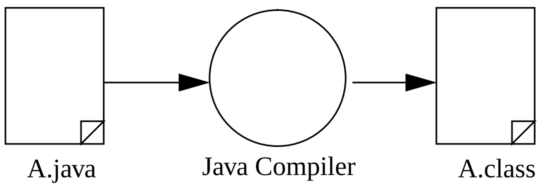


**Figure 4.1.** Generation of Java code

A code unit in Java is generated by a Java compiler, which translates Java classes and interfaces into equivalent bytecode units. These bytecode units then can be executed by a bytecode interpreter or further compiled into machine code by a Just-In-Time compiler. Code units consist of sets of class and interface definitions with at most one public class or interface.

## 4.3 Installation and initial transport of code

Whenever the Java interpreter has to use a class or interface for the first time since it was started (i.e. when an object of this class is instantiated), the interpreter is either looking for a corresponding class file on disk or asks a ClassLoader for the class. Our initial version of the class loading mechanism used an approach where the agent system ensured that the interpreter always finds a corresponding class on the file system. For that purpose, a component of the agent system, the Engine is asking another component, the CodeServer for the class of a migrating agent (1). The CodeServer decides whether the class already is loaded or not. In the latter case, the CodeServer is requesting the class from the Engine that sended the agent using HTTP (2). The file is then stored in the file system (3).

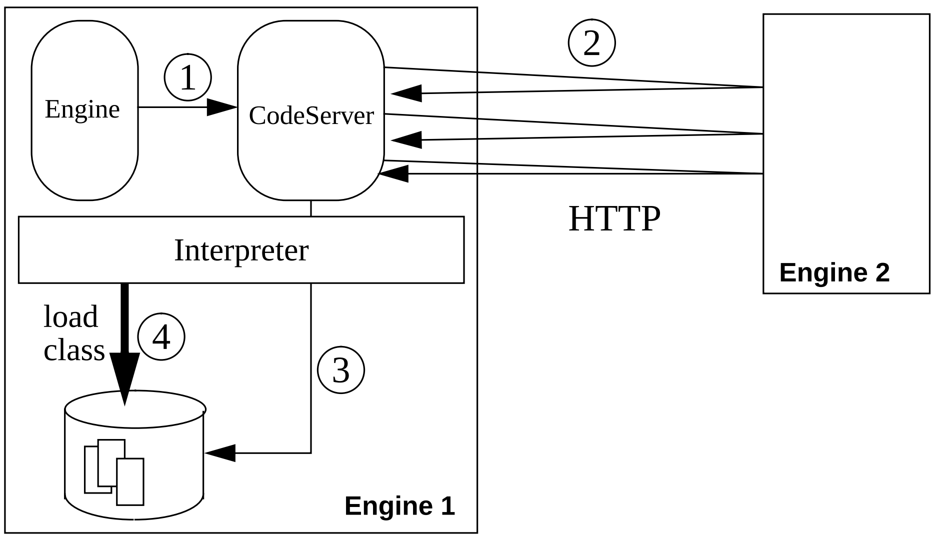


**Figure 4.2.** Inital code transfer and install mechanism

Since this new class may reference other, new classes, the CodeServer examines the requested class for such class references and requests these classes again from the other Engine. This process is executed until all referenced classes are stored in the file system. The request of the Engine (1) returns, and the Interpreter is now able to access all the class files it needs from the file system (4).

Although this first approach allowed us to transport code using a mechanism which is very similar to the one used in Java Applet systems, we decided to design and implement an own mechanism since the efficiency of the code transport is very important for the overall performance of the agent migration and since the Applet mechanism was not designed to be efficient. The results of our work are presented in Section 4.8.

## 4.4 Data representation

It exists a de-facto standardized representation of arbitrary Java objects, which is part of the Object Serialization package of JavaSoft. The format seems not to be documented currently, but since we do not write and read this format manually, but use the serialization API, this does not matter at the moment.

## 4.5 Generation of data

Serialized streams of Java Objects can be generated by the Serialization package, which is available for JDK 1.0.2 for several platforms. It is also (slightly modified) part of Java 1.1, so it should be available with modern JDK implementations.

## 4.6 Installation and initial transport of data

The “installation”, i.e. the injection of serialized Java objects in a running VM is also done by the serialization package.

The transport of data is very easy since the serialized agent object (which consists of the transitive closure of objects bound to instance variables) is send from the original system node to the receiving node via a TCP connection. At the receiving site, the agent then is de-serialized and installed.

## 4.7 Programming model

To offer Remote Execution and Weak Migration to the programmer, we had to design and implement a framework for the agents. Aspects of this framework are:
- naming of the receiving nodes
- execution model of the code
- programmer’s API

The naming aspects can be found in Section 3 . The execution model consists of five possible procedures, which are implemented by the agent, and called at different times by the system:

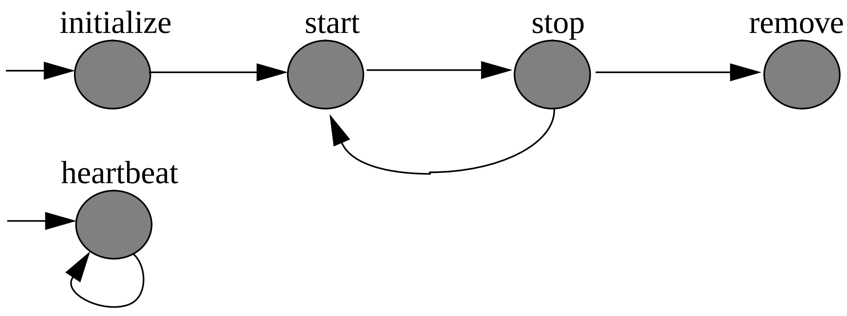


**Figure 4.3.** Execution model

- *initialize* is called, when an agent arrives at a location. It is not allowed to communicate with other agents in this procedure. In this procedure the agent can initialize variables, build up data structures, in short: prepare the execution.
- *start* is called after initialize to start the processing at the remote site. It is the “main” method of an Mole agent.
- *stop* is called, when the agent has to be stopped by the system e.g. before a migration of the agent. In this method, the agent can clean up e.g. by stopping threads or deleting data. It also can e.g. inform its owner.
- *remove* is called before the agent is removed from the location without the system wanting to restart the agent at a later point of time. Therefore the agent gets a last chance to e.g. contact its owner.
- *heartbeat* is called periodically when the agent registered for this service. It allows the periodical execution without the need of threads employed by the agent for this purpose.

### 4.7.1 Remote Execution API

To send, execute and control code at a remote location a programmer can use the following commands:

- `AgentHandle Location.sendAgent(MobileAgent, Locationname)`

  This method sends the agent MobileAgent to a location where it is started using the start() method. MobileAgent is an already instantiated agent object, but must not be an active agent known at the current location. The usage of an object reference instead of a class name allows the programmer to parametrize the agent by using standard (arbitrary) Java objects when calling the creation method of the agent. The method returns an AgentHandle, a ticket hat proves the “ownership” of the agent. This ticket can be used e.g. to delete the agent remotely. The method returns as soon as the agent is arrived at the remote location and ready to receive e.g. RPCs. This means, that the programmer can use the Mole RPC mechanism after having called sendAgent, a combination which is equal to a procedural remote procedure call with code transport.

- `boolean Location.deleteAgent(AgentHandle, Locationname)`

  This method allows to remotely delete an agent if the location and the AgentHandle is known. The result is true if the deletion was successfull and false else.

### 4.7.2 Weak Migration API

- `public void Agent.goTo(LocationName place)`

  This method allows an agent to migrate to Location `place`. In contrast to a strong migration, where the complete state of the agent (code, data and execution state)would be transported to the destination location, the agent is stopped and only the data of the objects belonging to the agent and its code are transported to the destination (weak migration).There, the agent is started using the start() mathod.

- `public void Location.createAgent(Agent anAgent)`

  An agent can be created (or better: installed at the agent system) by using this method. `anAgent` is a reference to the instantiated agent object. The system takes this agent, gives it a name and installs it at the current location.

- `public void Location.die(Agent theAgent)`

  If an agent want to be deleted, it call this method with a reference to itself as a parameter.

## 4.8 Efficient Code Migration for Modular Mobile Agents

The following paper is the result of our research in the field of efficient code migration. It was accepted and presented at the Third ECOOP Workshop on Mobile Object Systems: Operating System support for Mobile Object Systems in Jyväskylä, Finland, June 9-10, 1997. It will be published as part of the proceedings of this workshop. Although it speaks of mobile agents, the results are also applicable to Remote Execution code and even Java Applets.

### 4.8.1 Introduction

Mobile agents are groups of executing objects that can migrate as a whole from node to node in a heterogeneous network. To some applications, mobile agent systems offer advantages such as better performance, lower usage of network bandwidth and asynchronous processing. The migration of agents comprises the transport of data, code and execution state from one node to another. The transportation of data values is common practice in traditional distributed systems. The extension of an existing system service for the transport of data values into a service for code migration seems to be possible in a straightforward way. A closer look at the code migration component shows that its performance heavily influences the performance of the overall mobile agent system application and should therefore be optimized carefully. Despite the importance of efficient code migration, existing mobile agent systems make no use of this aspect besides the bare functionality of getting code somehow. The optimization of code migration is not trivial since for the migration of mobile agents in modular systems, not a single monolithic piece of code has to be transported, but a large set of classes that depend on another in various ways. An efficient implementation of code migration therefore has to exploit the properties of the underlying mobile agent system.

In this paper, we discuss code migration for mobile agent systems that use a modular code structure, e.g. an object-oriented approach. Most of this work also applies for the efficient migration of code of general mobile object systems.

The remainder of this paper is organized as follows: Section 4.8.2 lists the requirements of code migration in mobile agent systems. Section 4.8.3 describes the code migration model of the Java applet system, which is widely used in Java-enabled WWW browsers. In Section 4.8.4, a model of code migration is presented, that better fits the requirements. Section 4.8.5 discusses possible architectures of a code migrating component, the codeserver. The paper closes with a conclusion, some remarks about future work and a literature list.

### 4.8.2 Requirements of code migration in mobile agent systems

In order to find a satisfying code migration mechanism, we first have to state the requirements of mobile agent systems. These requirements are:

*there is a code source for any given class*

given a reference to a class, it has to be clear at any time on any node from which source this class can be requested

*code can be loaded at least when:*

- an agent migrates
- an agent interacts with a party on another node

Another requirement that does not need to be fulfilled in mobile agent systems, but provides a greater flexibility for programming could be:

- an agent wants to load a new class explicitly

*the code migration mechanism is robust, i.e. it works even in case of single node and temporary network failures*

i.e. single points of failures must not occur

*class references designate code versions known to the programmer*

since mobile agent systems can span a very large area with regard to the number of nodes as well as the time the system is running, it is not unusual that code is modified by the author in a manner that changes the behavior of the code, although not all usages of this class can reflect this change. This means that all versions of a class that may be used by other code have to be stored and that references designate the expected version of a class

*code is protected against modification attacks like viruses and trojan horses*
when code is not transported in one piece, but contains a list of class references that are used to find the correspondent classes, attacks can use modified versions of a class e.g. by inserting "intrusion code"

*code is migrated in an efficient manner*
since code migration represents an essential part of the overall performance of the agent migration, the code migration component has to perform well in order to allow the mobile agent system to perform well

*the code migration mechanism fits into the asynchronous processing model that is offered by mobile agents*
one advantage of the mobile agent model is the ability to send asynchronous jobs by using mobile agents instead of maintaining a connection between client and server. In order to preserve this advantage, the code migration mechanism has to cope with agent nodes, that are is not reachable during all the processing of the agent

*the code migration mechanism fits into the organizational model of the mobile agent system*
e.g. if we foresee an organizational model that employs an 'egocentric' approach of the participating institutions, we cannot successfully use a code migration mechanism that relies on a "selfless acting" mobile agent infrastructure.

Before we design a code migration component, we should examine mechanisms that also transport code, especially the Java applet code model.

### 4.8.3 Related Work

The problem of code transport is common not only to mobile agent systems, but also to some of the mobile object and to mobile code systems. At current, they can be divided, according to their code requesting method, in two groups:
1. systems that view code as one piece of code and which ship it as a whole
2. systems that use the Java applet code requesting method

Systems of the first groups are e.g. ffMAIN [LDD95], Messengers [Tsc94], Tacoma [JRS95] or Ara [Pei96], systems of the second one e.g. Aglets [IBM96] or Odyssey [GM97]. To our knowledge, none of the current systems deals with efficiency aspects of transporting code.

Although the Java applet code requesting method does not try to be efficient or fault-tolerant, it enables an agent system to see the code of an agent as a set of classes that can be transported separately. Therefore, we will now present the code model of the Java applet system, further we will show, that it is not adequate for efficient code migration.

#### The Java Applet Code Model

Figure 4.4. presents the mechanisms that are used to get new code into a Java-enabled browser: In the code of a class A, an object of class B has to be created. Since class B is not found in the set of loaded classes, the java interpreter asks a system component, the *ClassLoader* for this class. Depending on the source of class A, the ClassLoader tries to get the code for class B from the same source. When A was loaded from the local file system, the ClassLoader is looking there for B, when A was loaded via HTTP from a WWW server, it tries to get class B from the same WWW server using the same URL prefix. Once the ClassLoader gets the class file, it initializes class B, which then can be used by the interpreter to create an instance of it.

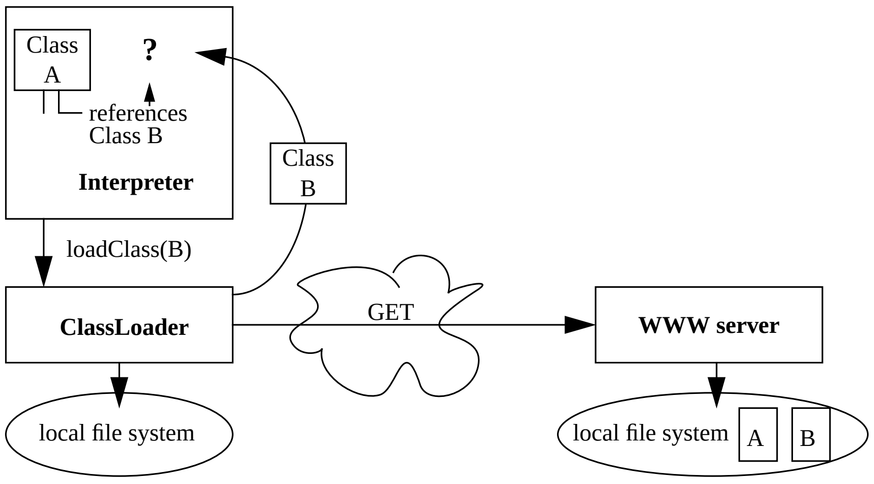


**Figure 4.4.** Java Applet Code Model

If we now look on whether this mechanism satisfies the requirements for a mobile agent system, we find some deficiencies: There exists a way to determine the source to a given class, but noone can guarantee that this source really holds this class. Since only a single source is used, code cannot be loaded in case of a failure of the source. Classes are identified solely by their names, and there are neither mechanisms to distinct different class versions nor different classes that just share the same class name (there are some mechanisms in newer Java versions, but they cover only parts of the problem). Since the classes of one applet are loaded from only one source, one could argue, that it is impossible to inject malicious code, and there are mechanisms in Java 1.1 that allow to sign class archives, but as soon as an applet loads code from more than one archive, this protection can be broken. The efficiency of the code migration depends in the Java applet system on the performance of the transport protocol (HTTP) and on the actual performance of the WWW server used as the source. Since the network performance of the source to the applet receiver is random (there is no relationship between a source and the network location of the receiver of an applet), and since applet are offered often by few or even one source, the performance is often poor. Finally the load-on-demand characteristic interferes with the requirement of asynchronous interaction since classes can be load even at the end of the lifetime of an applet.

Although the above code migration mechanism is adequate for its purpose in the Java applet system, it does not match that good to the needs of code migration for mobile object and mobile agent systems. Therefore, we will follow an approach that fits better our requirements.

### 4.8.4 The Codeserver Model

In this section, we will present the model of a code migration mechanism that is able to satisfy the requirements of code migration in mobile agent systems.

The code migration component, the *codeserver,* is requested for one or more classes by an interpreter when one of three events occurs:

- a new agent arrives
- an agent receives a communication request with parameters of new classes
- an agent requests loading a new class

where “new class” means one that does not exist in the local class storage of the corresponding codeserver. In all cases, the interpreter has a reference to only one class, i.e. one that is specified by the arrived instance (or by a class name). The need for other classes then results from the “main” class using instances of other classes. The codeserver is able to find the class code at another codeserver and loads the code into its code storage, from where it can be loaded into the interpreter. Since this class may reference other new classes, the whole mechanism continues until no more new classes are required.

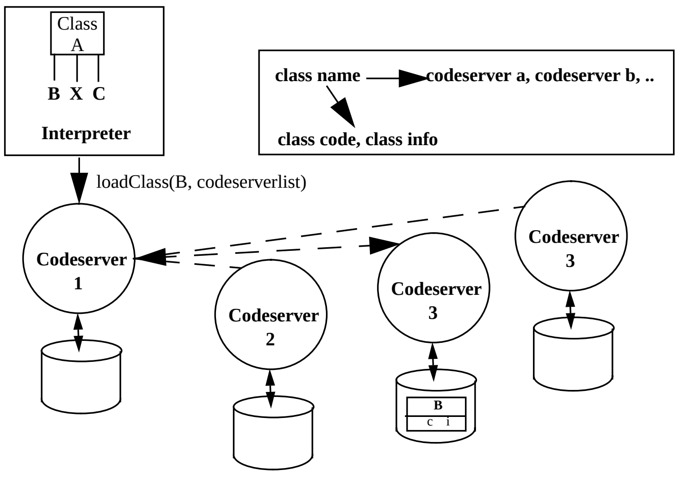


**Figure 4.5.** Code server model

It is interesting to examine the class requests by one class over its lifetime. Some classes are referenced by their name in the code of the main class, but it is not sure that all of them are ever used (e.g. when a variable is never instantiated). If the agent communicates, it gets sometimes data elements that are instances of new classes. Although some of the superclasses of these objects are known in advance in inheritance oriented languages (since they are also part of variable declarations), the current classes of the communicated instances may be new. The third group of classes is referenced also by their name, but not as part of the programming language, but as strings. Therefore, the system cannot determine these class names at startup time of the agent, but has to wait for the explicit class request by the agent.

Every interpreter needs one codeserver that can supply classes when needed, but a codeserver may serve more than one interpreter. The codeserver is a self-containing component that can be located at the same computer like an interpreter, but it does not have to. Normally, a codeserver is located “near” the interpreters it serves, e.g. in the same LAN. An interpreter can find a codeserver either by asking other interpreters or by reading configuration data provided at installation time (since this is comparable to finding the e.g. DNS-Server of a network). A codeserver knows other codeservers in its “neighborhood”, i.e. such, that are reachable by good connections, but sometimes, it may also contact codeservers, that are far away. A codeserver has only a limited code storage, therefore, it has to employ class replacement mechanisms in case it is low on memory.

After defining the model, we will now examine the ways we can implement this model in an efficient way.

### 4.8.5 Codeserver Architectures

We will now present the techniques that are needed to implement the above model. First, we will show the two mechanisms that are used to get a class given its name, then we will discuss the details of the code transport and the interfaces to the codeserver. Finally, we will present a code replacement mechanism that is able to take connection characteristics into account.

When we look at the requirements, we find that, on the one side, a class has to be found even in case of node failures, and, on the other side, the code migration mechanism has to be as efficient as possible. The first requirement either needs a static association of a class name to a set of given codeservers that are "responsible" for providing that class, or it needs a mechanism that is able to locate any class among the codeservers. In each case, at least one server has to provide this class. The second requirement should use the fact, that some of the other codeservers are "easier" to reach than others, i.e. that the transport of a class is faster. Since the mechanisms that are needed for both requirements, do not influence each other, we suggest to use two different mechanisms for the two different purposes. The *standard mechanism* tries to get code transported as efficient as possible, but may fail sometimes in finding a class. The *basic mechanism* is able to get any class, but will not do this in a fast way. Therefore, we use normally the standard mechanism and switch back to the basic mechanism if the first one fails.

#### The basic mechanism

The purpose of the basic mechanism is to get any class given a class name somehow, i.e. without the requirement of efficiency. As the basic mechanism, any method is adequate that is able to associate a code server to a class name at any time in a way, that a) the code server has this class, b) the codeserver is reachable by the interpreter and c) that fits into the organizational model of an open mobile agent system without a central infrastructure. "At any time" means in this context, that the method has to be able to cope with single code server failures, e.g. by the use of replication of classes. There are some existing components that offer this functionality, e.g. distributed file systems with replication facilities like DFS [KLA90] or general object location systems (code pieces are in this sense "objects") like the one used in Hermes [BA90]. Since this aspect is not so important to efficiency, we decided to implement an own, small mechanism that also offers the requested features and fits into our architecture without the need of using large subsystems that are not that portable like the rest of the agent system.

In our basic mechanism, in order to find a codeserver that is "responsible" for a certain class, classes have to be registered by the programmer at a codeserver before they can be used in the agent system. For making it easier to find the "home server" of a class, the name of this server becomes part of the class name. At the same time, this registration allows unique class names since the name of the registration unit is a part of the name, and since this unit may also ensure the uniqueness of the original class name in its "domain".

One requirement of mobile object systems was that references to different code versions should refer to different classes. Since we now have a mechanism that register classes, it can also insert iterated version numbers into the class name, making sure that different class versions can be distinguished. In the class itself, the extended class name can be inserted automatically.

The described mechanism allows us to determine one server that holds a certain class. Since the failure of this server would lead to a failure of the basic mechanism for this class with respect to the proposed solution the degree of fault tolerance needs to be enhanced . Therefore, we allow not only one *primary* server for this class, but also some *secondary* servers. The secondary servers also store this class, they do not occur as a part of the class name, but are provided as additional information.

Whenever a codeserver has to load a class, and is not able to use the standard mechanism, it extracts the name of the primary and secondary servers from the class name and the associated informations, and asks these servers until one of them is available and can, therefore, provide the code.

### The standard mechanism

The purpose of the standard mechanism is to retrieve new classes as fast as possible. This aspect does currently not occur in systems that migrate code. Therefore, new mechanisms have to be developed. Our approach tries to improve efficiency by fetching code fragments from other, “neighboring” codeservers. The neighbor relationship is expressed through the codeservers’ distance. Therefore, two things have to be computed: the set of possibly neighboring codeservers, and the real “distance” to them both in terms of roundtrip delay and transmission speed.

The detection of neighbor candidates can be done in several ways: by an explicit configuration, by a codeserver directory service, by exploitation of the addresses of the primary and secondary codeserver of the stored classes, or by an examination of the interaction partners of agent system nodes plus the information which codeserver provides code to that nodes.

The “distance” to the set of found candidates can then be measured by the usual means; the best n candidates are then neighbors and will be taken in the future into account.

In order to further speed up the mechanism, neighbors exchange lists of stored classes, so a codeserver does not have to first ask around for a class. For not exchanging whole class lists, normally only updates of the lists are exchanged.

An additional way to make the code transport faster is to prefetch classes that may be used in the future. For this purpose, a class provides additional information about which classes in the past were loaded how often by requests coming from this “main”-class. The codeserver then can get at least the most frequent classes in advance. If the behavior in the past allows to foresee the future class references, this mechanism allows to load classes that not even occur in the code of the main class. This means also, that the loading policy is not on-demand, but greedy in a way that the m most frequent referenced classes are loaded while the main class is running. This and other mechanism that try to exploit informations about the (potential) behavior of interpreters is the cause for allowing the interpreter to use only one codeserver. If it could use at least one several codeservers would try to prefetch code in parallel, which is inefficient without synchronization. Further optimization could exploit the fact, that it is more efficient to get more classes instead of one from one partner, since the overhead to establish a connection is the same.

Beneath the question of how code servers are found, the transport itself has to be examined.

## Code transport

Code has to be transported between two codeservers, and between the codeserver and the requesting interpreter. Since classes have to be registered, code also has to be exchanged between a programmer and a codeserver. Each of these exchanges requires an interface.

**Interfaces**
We have to define three interfaces (described here in pseudocode):

**Interpreter - Codeserver**
`Class getClass(Classname)`
This method is used by the interpreter to request a class from the codeserver by its name.
`Address[] getCommunicationPartners()`
The codeservers uses this method to get the known communication partners (i.e. interpreters) from the interpreters it provides with code.
`Address getCodeserver()`
This method is used by codeservers to request the name of the (mainly) used codeserver of this interpreter. This method can be used to find new neighbor candidates.

**Codeserver - Codeserver**
`Class getClass(Classname)`
This method is used to request a class by its name.
`Classname[] getDirectory()`
This method is used to request the entire list of classes stored in a codeserver.
`Classname[] getDirectoryDelta()`
This method is used to get an update of the list of classes stored in or deleted from a codeserver with relation to the last directory request.

**Codeserver - Programmer**
`Classname registerClass(Class, Classname)`
The programmer uses this method to register a class at a given primary server.
Since these interfaces interact between different parties, also the implementations of the interfaces is different.

**Transport protocols**
For the “physical” transport of code between two entities, any existing efficient file transfer protocol can be used. These protocols include long-existing and well-proofed standards like FTP, HTTP or even SMTP. Another interesting candidate is WebNFS [Cal97], a version of NFS that was designed to allow web browsers and Java applets to get access to NFS servers even through corporate firewalls.

Together with the two code obtaining mechanism we now have a service that allows us to transfer any registered class onto an interpreter. What we have to consider now is the storage limitations of a codeserver, i.e. the question of what to do when a new class is needed, but the storage space is low.

## Code replacement policies

When a new class has to be loaded, but the storage space is low, code replacement policies can be employed to find stored classes that can be removed. These policies have to consider the fact that a single codeserver serves multiple interpreters and the “class garbage policy” of the interpreters. If the interpreters do not remove classes in use, the codeserver can make less assumptions about whether classes are needed in the future. Therefore, the easiest code replacement policy is called *Random*: in case of the need of a replacement, a random class is removed. This policy is based on the assumption, that the classes are requested by the interpreters independently and that an interpreter does not remove a class until no instance references it anymore. If an interpreter is able to remove even classes currently in use (but not active for a while), it can signal such a removal to the codeserver which then assumes, that a request of this class is probable in the future.

Efficient code replacement policies have to assume a certain class request structure. The random policy for example assumes an equal distribution of the class requests. Another policy, which is called Least Recently Used (LRU, see [Tan92]), assumes that classes that have been requested recently, will also be requested in the future and that this probability diminishes with the length of the interval of the last access. This can be true for a single codeserver, e.g. when the provided interpreters relate in a manner that it is probable, that an agent migrates from one of these interpreters to another of this group. Of course, this does not have to be the case, but the question of how the “class request profile” is structured, is application dependent, and cannot measured today since there are few real world applications that use mobile agents yet.

In contrast to memory replacement policies known from the field of operating systems, the costs of reloading a removed class can be different depending on the distance of the reloading source. Therefore, not only the probability of whether a class will be requested in the future is important, but also these costs have to be considered. For computing the reloading costs, the list of stored classes must be known, and the distance to the potential sources of these classes (which is - for the neighbor servers - already the case for the standard algorithm).

What we have to consider is that prefetched classes (compare Section 4.6.2 ) that are not already requested by an interpreter, will be probably needed in the near future. Therefore the replacing algorithm should give these classes more time before they will be replaced (with the same argument as above, the interpreters should inform the codeservers about classes that have been garbage collected).

A good starting point of the code replacement policy of a codeserver is, therefore, a Random or LRU algorithm that also considers the cost of reloading a class and that considers prefetched classes as described above.

The final aspect to be examined is the question how to protect code against modification attacks.

### Security aspects

Without code protection, an attacker can use the codeserver mechanism to comfortably inject malicious code into agent classes, and therefore, agent applications. All it has to do is to modify passing classes either by attacking through the network or by opening an own codeserver. Therefore, registered code has to be protected against modification attacks. This can be achieved through the usage of digital signature techniques known from the field of cryptography. These techniques, e.g. DSS (Digital Signature Standard, see e.g. [Sta95] for details) allow to create an unforgable signature that matches exactly one document. What we have to do here, is to employ such an algorithm, letting the programmer sign its classes (by using its secret key) and making its public key available in a secure way (see e.g. [Sta95] for a discussion of how to spread public keys). Finally, we have to associate the signature of a class to the reference to that class, i.e. the programmer of an agent does not just need to name the class it want to use, but also its signature. The question of how a programmer is building up trust into the classes it uses, is important, but not an aspect of the migration of code and will be omitted here. Instead of using the programmers key, any key of a trustworthy institution can be used for purpose of signing code.

## 4.9 Conclusions and future work

In this section, we presented an execution framework for Java agents. The mechanisms providing remote execution and weak migration are mostly based on standard java mechanisms. While the Java object serialization is well suited for the transport of agent data and the transport format for the agent code is determined by the Java byte code, we investigated some additional research in the efficient distribution of the agent code.

As a result, we presented a code migrating mechanism that fulfills the requirements of mobile agent systems:

- for any class, the component, which is called codeserver, is able to locate a server that is responsible providing the code of this class
- code can be loaded anytime, even by the explicit request of a class
- the mechanism is robust against the failure of single codeservers
- different versions of the “same” code can be differentiated and the codeserver tries to get the version that is expected by the programmer of a class
- code is protected against modification attacks by the usage of digital signatures
- the “standard mechanism” for obtaining code is designed to work in an efficient manner
- the distinction between a code serving component and the agent runtime environment allows to send away agents without the need for the sending node of holding a connection to the network for a longer time, allowing therefore an asynchronous application model
- a node of the agent system needs a codeserver in order to work. This codeserver either has to be maintained by the node, or it may use any already running instance

The described mechanism has been currently implemented (see [Kla97] for details) in our mobile agent system, Mole [SBH97], which is build upon the Java language [GJS96]. We expect a significant performance increase of the agent migration by using a codeserver, and we will, therefore, compare this potential increase by measuring the migration times in agent applications such as active documents or network management. Other positive effects of using such a code migration component is the reduction of the size of an agent system node, which allows the usage of the system even in smaller devices, and the solution of some system problems such as versioning of classes and the protection of code. Finally, we will try to find further optimization mechanisms like compression of classes by examining code request profiles of different applications.

# 5 Performance Model and Verification

It is often argued that the advantage of agent migration lies in the reduction of (expensive) global communication costs by moving the computation to the data ([HCK95][GM96]). Although this argument is understandable from an intuitive point of view, not much work has yet been done to evaluate the performance of migration on a quantitative basis. A simple performance model was investigated in [CPV97]. The evaluation of three scenarios was done in [CK97].

In this section a performance model regarding network load and execution time is developed, which can help to identify situations for which agent migration is advantageous compared to remote procedure calls. This performance model is intended to help to decide which interaction model to be used in different scenarios of a mobile computation environment.

## 5.1 Interaction Models

Interaction between entities (objects, agents) in distributed systems which are located at different sites can take place in many different ways ([BH+97]). In this paper we will confine on the remote procedure call as a global communication mechanism on one side and on agent migration via the network to the communication partner followed by local communication on the other side.

### 5.1.1 Remote Procedure Calls (RPC)

Remote Procedure Calls are a widely used interaction mechanism in distributed systems. Basically a procedure is executed on a remote server, transferring the control flow (including some arguments) from the client to the server until the request is executed and the result is returned ([BN84]). Extensions to this synchronous concept include, among others, asynchronous RPC.

### 5.1.2 Agent Migration

Agent migration is a mechanism to continue the execution of an agent on another location ([GM96]). It includes the transport of agent code, execution state and data of the agent. In an agent system, the migration is initiated on behalf of the agent and not by the system (e.g. for load balancing purposes). The basic motivation for migration is to move the computation to a data server or a communication partner in order to reduce network load by accessing a data server or communication partner by local communication.

### 5.1.3 Remote Execution

Remote execution in the context of mobile agents is a mechanism to start (rather than to continue) the execution of an agent on another location. It includes only the transport of agent code and some parameters. Due to the similarity of remote execution and agent migration regarding their communication needs, remote execution is not further considered in this paper.

## 5.2 A Single Interaction

In this section a single client/server-style interaction is considered. The following simplifying assumptions are made: The interaction partners and the amount of communication for request and reply in each interaction is known in advance. Average values for delay and throughput are considered. The time for marshalling and unmarshalling (i.e. transformation of entities in a transport format and back) increases linear with the size of the data to be sent. All locations execute jobs with the same speed.

### 5.2.1 Interaction by RPC

In the context of agent interaction, the RPC is used to call procedures (methods) that are provided by the communication partner (e.g. a service agent). A (classical) RPC includes binding to the server, marshalling, transfer, unmarshalling of the request parameters, execution of the request, and marshalling, transfer and unmarshalling of the reply. With the above mentioned simplifying assumptions, the time for binding can be omitted since communication partners are known in advance and the time for execution of the request can be omitted since not influenced by the interaction model. Marshalling is dependent of the size of the request parameters only. The performance model therefore concentrates on the communication dependent part of the RPC.

The network load $B_{RPC}$ (in bytes) for a simple remote procedure call from location $L_1$ to location $L_2$ consists of the size of the request $B_{req}$ and the size of the reply $B_{rep}$:

$$B_{RPC}(L_1, L_2, B_{req}, B_{rep}) = \begin{cases} 0 & \text{if } L_1 = L_2 \\ B_{req} + B_{rep} & \text{else} \end{cases}$$

The execution time $T_{RPC}$ for a simple remote procedure call from location $L_1$ to location $L_2$ consists of the time for marshalling and unmarshalling of request and reply (factor $\mu$) plus the time for the transfer of the data on a network with throughput $\tau(L_1,L_2)$ and delay $\delta(L_1,L_2)$

$$T_{RPC}(L_1, L_2, B_{req}, B_{rep}) = 2\delta(L_1, L_2) + \left(\frac{1}{\tau(L_1, L_2)} + 2\mu\right)B_{RPC}(L_1, L_2, B_{req}, B_{rep})$$

### 5.2.2 Interaction by Agent Migration

In this section, an interaction is performed by the migration of the agent to the communication partner, a local or remote procedure call, the processing of the data received and the transfer of the processed data back to the source location. A (classical) migration includes marshalling, transport and unmarshalling of code, data and execution state of the agent to the server. With the same simplifying assumptions as above, the time for the execution of the procedure call can be omitted. Marshalling increases linear with the size of data and execution status of the agent, while the code of the agent is already stored in transport format and is only transferred on demand (i.e. if the code is not yet available at the server).The agent consists of $B_{code}$ bytes of code, $B_{data}$ bytes of data and $B_{state}$ bytes of execution state and is described by the triple $B_A$=($B_{code}$, $B_{data}$, $B_{state}$). The size of the request to the procedure $B_{req}$ is yet contained in $B_{data}$. The size of the reply from the procedure $B_{rep}$ is reduced by remote processing to $(1-\sigma)B_{rep}$ with $(0 \leq \sigma \leq 1)$ by the agent where $\sigma$ models the selectivity of the agent.

The network load for the migration of an agent $A$ from location $L_1$ to location $L_2$ is calculated by

$$B_{Mig}(L_1, L_2, B_A) = \begin{cases} 0 & \text{if } L_1 = L_2 \\ P(B_{cr} + B_{code}) + B_{data} + B_{state} & \text{else} \end{cases}$$

where $P$ denotes the probability that the code is not yet available at location $L_2$ and $B_{cr}$ is the size of the request from $L_2$ to $L_1$ to transfer the code. If the agent additionally sends back a reply message to location $L_1$, the network load amounts

$$B_{MR}(L_1, L_2, B_A,\sigma,B_{rep}) = B_{Mig}(L_1, L_2, B_A) + \begin{cases} 0 & \text{if } L_1 = L_2 \\ (1-\sigma)B_{rep} & \text{else} \end{cases}$$

where $B_{rep}$ is the size of the reply and $\sigma$ denotes the selectivity of the agent.

The corresponding execution time including delay $\delta$, throughput $\tau$ and marshalling overhead $\mu$ for a single agent migration from location $L_1$ to location $L_2$ is described by

$$T_{Mig}(L_1, L_2, B_A) = (1 + 2P)\delta(L_1, L_2)$$

$$+ \frac{B_{Mig}(L_1, L_2, B_A)}{\tau(L_1, L_2)} + \begin{cases} 0 & \text{if } L_1 = L_2 \\ 2\mu(B_{data} + B_{state}) & \text{else} \end{cases}$$

and including the reply message back to location $L_1$, the execution time amounts

$$T_{MR}(L_1, L_2, B_A,\sigma,B_{rep}) = T_{Mig}(L_1, L_2, B_A) + \delta(L_1, L_2)$$

$$+ \begin{cases} 0 & \text{if } L_1 = L_2 \\ \left(\frac{1}{\tau(L_1, L_2)} + 2\mu\right)(1-\sigma)B_{rep} & \text{else} \end{cases}$$

### 5.2.3 Evaluation of a Single Interaction

To evaluate a single remote procedure call and a single agent migration based on these simple models, we consider the following scenario: The agent consists of $B_{code}$=39kBytes of code, $B_{data}$= 5kBytes of data and $B_{state}$=5kBytes of execution state. With a probability of $P$=10% the code of the agent is not yet available at the remote location, in this case the transmission of the code has to be initiated by a code request message of size $B_{cr}$=1kByte. The request size of the interaction is $B_{req}$=1kByte. Figure 5.1. compares the network load of the remote procedure call

($B_{RPC}$) to agent migration with reply ($B_{MR}$) for a fixed reply size of $B_{rep}$=25kBytes while varying the selectivity σ between 0% and 100%.

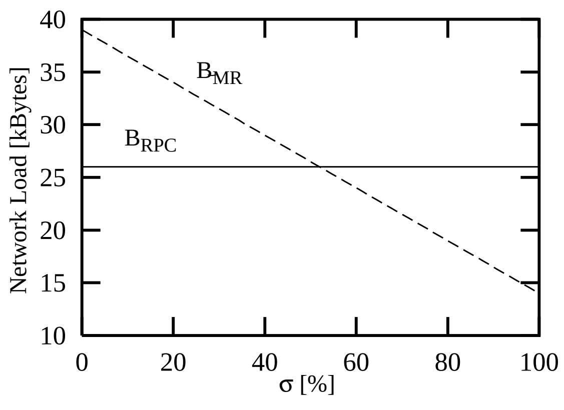


**Figure 5.1.** Network load versus selectivity

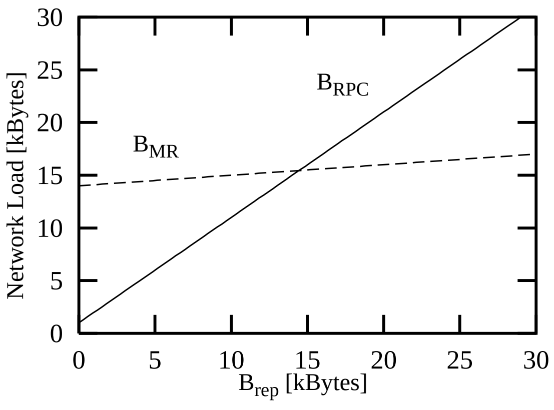


**Figure 5.2.** Network load versus reply size

Figure 5.2. compares the network load of remote procedure call ($B_{RPC}$) and agent migration with reply ($B_{MR}$) for a fixed selectivity of σ=0.9 while the reply size $B_{rep}$ is varied between 0 and 30kBytes.

The diagrams in Figure 5.1. and Figure 5.2. show that the usage of agent migration rather than the usage of remote procedure calls reduces network load only if the size of the reply is large and/or the agent has a large selectivity.

Figure 5.3. and Figure 5.4.show the corresponding diagrams for the execution time with assumed network characteristics delay δ=30ms, throughput τ=400kBytes/s and no marshalling overhead (μ=0s/kByte).

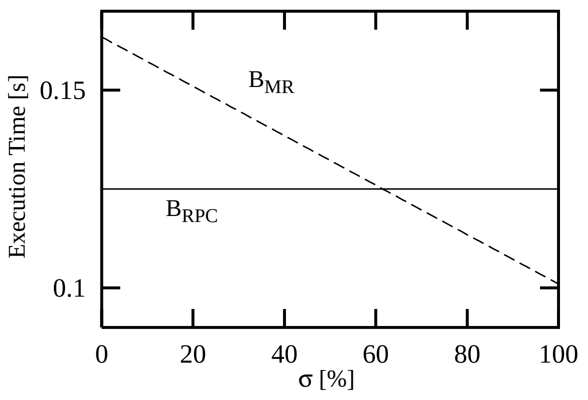


**Figure 5.3.** Execution time versus selectivity

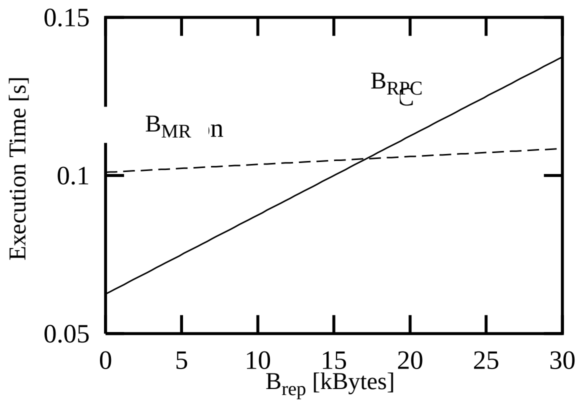


**Figure 5.4.** Execution time versus reply size

While the overall behaviour of these graphs imply the same conclusions, it should be noted that the break-even point between remote procedure call and agent migration for network load differs from the break-even point for execution time. For example, the break-even point for network load in Figure 5.1. is reached at σ=52%, while the break-even point for execution time in Figure 5.3. is reached at σ=62%.

## 5.3 A Sequence of Interactions

In this section a sequence of several interactions with different interaction partners on different locations is considered. The following simplifying assumptions are made: The sequence of interaction partners and their locations, as well as each request size, reply size, selectivity and number of communications per location is known in advance. Furthermore it is assumed that average delays and average throughput in a possibly inhomogeneous network are known for every possible interconnection.

Let $S=(I_1,...,I_n)$ be a sequence of interactions. The $i$-th interaction is described by

$$I_i = \{R_i, m_i, B_{req_i}, B_{rep_i}, \sigma_i\}$$

where $R_i$ is the remote location with which the communication should take place. Each communication consists of $m_i$ (local or remote) procedure calls with request size $B_{req_i}$, reply size $B_{rep_i}$ and selectivity $\sigma_i$. The size of the agent after interaction $i$ is modelled for i=0,...,n by

$$B_{A_i} = (B_{code_i}, B_{data_i}, B_{state_i})$$

where $B_{code_i}$ and $B_{state_i}$ remain fixed while

$$B_{data_i} = B_{data_{i-1}} + m_i(1-\sigma_i)B_{rep_i}$$

The mobility behaviour of the agent is described by a destination vector $D=(D_0,...,D_n)$. For the $i$-th interaction the agent moves to destination location $D_i$. Thus migration takes place between $(i$-$1)$-th and $i$-th interaction only if $D_i \neq D_{i-1}$. Please note that the agent need not migrate to location $R_i$ where the next interaction takes place.

The network load and execution time for a mixed sequence of remote procedure calls and agent migrations are calculated as follows

$$B_{Seq}(S, D, B_{A_0}) = \sum_{i=1}^{n} (B_{Mig}(D_{i-1}, D_i, B_{A_{i-1}}) + m_i B_{RPC}(D_i, R_i, B_{req_i}, B_{rep_i}))$$

$$T_{Seq}(S, D, B_{A_0}) = \sum_{i=1}^{n} (T_{Mig}(D_{i-q}, D_i, B_{A_{i-1}}) + m_i T_{RPC}(D_i, R_i, B_{req_i}, B_{rep_i}))$$

### 5.3.1 Evaluation of Scenario 1

To evaluate a sequence of interactions based on this performance model, we considered a typical situation in mobile computing, namely the usage of a laptop or personal digital assistant (PDA) at location $L_0$ that only has wireless low bandwidth access to the Internet. The characteristics of this rather inhomogeneous network are assumed as follows: Each interaction between locations x and y inside the Internet has a delay of $\delta(x,y)$=10ms and a throughput of $\tau(x,y)$= 400kBytes/

s. Each interaction between the PDA and the Internet has a delay of δ(x,y)=120ms and a throughput of τ(x,y)=50kBytes/s.

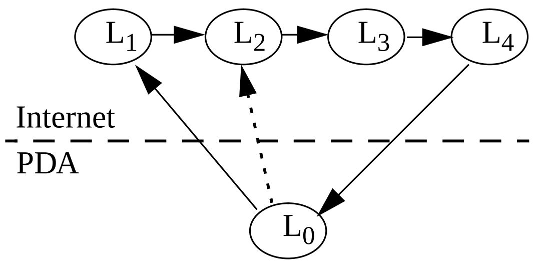


**Figure 5.5.** Mobile computing scenario

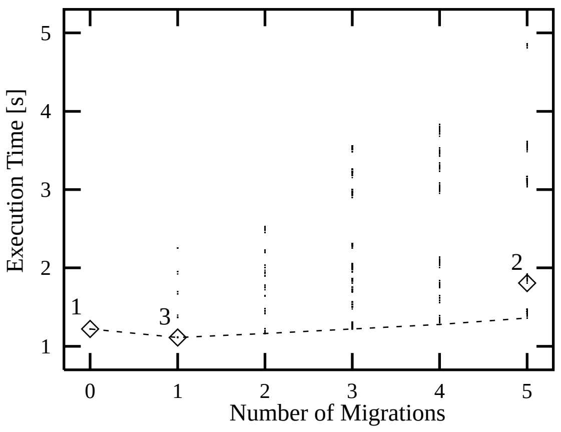


**Figure 5.6.** Execution times for scenario 1

The mobile agent starts on the PDA at location $L_0$ and has to interact with four locations $L_1$...$L_4$ before returning the result to $L_0$ (as illustrated by thin arrows in Figure 5.5.).

The amount of communication with locations $L_2$ and $L_4$ is much larger than with the other locations. In particular, in scenario 1 a mobile agent with $B_{code}$=10kBytes, $B_{data}$=5kBytes, $B_{state}$=5kBytes is considered. It is assumed that the code of the agent is not available at the remote locations ($P$=100%) and that the size of the agent's data does not increase (σ=1). The mission of the agent is to process the sequence of interactions $S_1$ listed in Table 1.

Table 1: Sequence of interactions $S_1$

| $i$ | $R_i$ | $m_i$ | $B_{req_i}$ | $B_{rep_i}$ | $\sigma_i$ |
|---|---|---|---|---|---|
| 1 | $L_1$ | 1 | 50 | 2000 | 1 |
| 2 | $L_2$ | 1 | 500 | 4000 | 1 |
| 3 | $L_3$ | 1 | 50 | 2000 | 1 |
| 4 | $L_4$ | 1 | 500 | 4000 | 1 |
| 5 | $L_0$ | 1 | 500 | 10 | 1 |

The performance model for interaction sequences was now used to evaluate the execution time for all possible agents regarding their mobility behaviour. The diagram in Figure 5.6. shows the execution time for all possible destination vectors $D$ sorted on the horizontal axis according to the number of migrations involved.

Three of the execution times are marked in the diagram and the corresponding values of the destination vector $D$, the execution time $T_{Seq}$ and the network load $B_{Seq}$ are listed in Table 2 for further inspection. Evaluation 1 indicates the execution time of an agent that only uses RPC and no migration at all. Evaluation 2 on the other hand uses migration to the location of the next interaction partner for each of the five interactions. The execution time is much larger than for the RPC-only evaluation 1 due to the much larger network load involved. Figure 5.6. furthermore shows that evaluation 2 is not the fastest solution with five migrations. Due to the low through-

Table 2: Performance values for scenario 1

| # | Destin. Vector *D* | $T_{Seq}$ | $B_{Seq}$ |
|---|---|---|---|
| 1 | $L_0$,$L_0$,$L_0$,$L_0$,$L_0$, $L_0$ | 1.2220 | 13100 |
| 2 | $L_0$,$L_1$,$L_2$,$L_3$,$L_4$, $L_0$ | 1.8075 | 105000 |
| 3 | $L_0$,$L_2$,$L_2$,$L_2$,$L_2$, $L_2$ | 1.1117 | 30110 |

put of the wireless link to the PDA, it is still better to make a (useless) fifth migration to any other internet location ($L_1$ - $L_4$) before interaction $I_5$ and then to use an RPC to transmit the results back to location $L_0$. Interestingly, the shortest execution time is achieved with evaluation 3 by an agent that uses migration exactly once. The destination vector for evaluation 3 shows that the corresponding agent migrates to location $L_2$ before interaction $I_1$ (i.e. not to the interaction partner of $I_1$ but to the first location with a large amount of communication) as indicated by the dotted arrow in Figure 5.5. and remains there until to the end of the sequence. Again the network load is larger than for evaluation number 1, but the intelligent usage of a single migration reduces execution time compared to the RPC-only evaluation 1.

### 5.3.2 Evaluation of Scenario 2

In the second scenario the same values for the agent ($B_A$, $P$ and $\sigma$) are used, but the sequence of interactions changes slightly according to Table 3. In particular only interactions 2 and 4 are

Table 3: Sequence of interactions $S_2$

| $i$ | $R_i$ | $m_i$ | $B_{req_i}$ | $B_{rep_i}$ | $\sigma_i$ |
|---|---|---|---|---|---|
| 1 | $L_1$ | 1 | 50 | 2000 | 1 |
| 2 | $L_2$ | 10 | 50 | 400 | 1 |
| 3 | $L_3$ | 1 | 50 | 2000 | 1 |
| 4 | $L_4$ | 10 | 50 | 400 | 1 |
| 5 | $L_0$ | 1 | 500 | 10 | 1 |

modified by replacing one communication and large request/reply size with ten communications and small request/reply size, so that the total amount of data transferred is not changed.

The diagram in Figure 5.7. shows the evaluation of the execution time for all possible destination vectors, again sorted according to the number of migrations involved. The smallest execution time (evaluation number 4) is achieved by an agent that uses migration twice. Table 4 lists the corresponding values of the destination vector, execution time and network load.
Evaluation 1 shows the values for the agent using RPC only. Although the amount of data transmitted is the same as in scenario 1, the execution time increased heavily due to the larger number of communications in scenario 2 because each additional RPC increases the execution time by

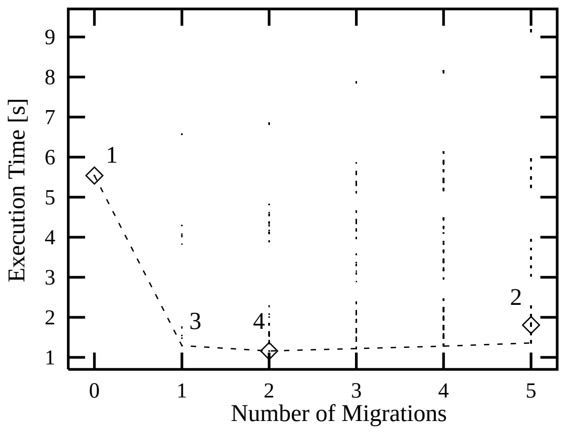


**Figure 5.7.** Execution times for scenario 2

Table 4: Performance values for scenario 2

| # | Destin. Vector $D$ | $T_{Seq}$ | $B_{Seq}$ |
|---|---|---|---|
| 1 | $L_0$,$L_0$,$L_0$,$L_0$,$L_0$,$L_0$ | 5.5420 | 13100 |
| 2 | $L_0$,$L_1$,$L_2$,$L_3$,$L_4$,$L_0$ | 1.8075 | 105000 |
| 3 | $L_0$,$L_2$,$L_2$,$L_2$,$L_2$,$L_2$ | 1.2917 | 30110 |
| 4 | $L_0$,$L_2$,$L_2$,$L_2$,$L_4$,$L_4$ | 1.1629 | 46610 |

two delays. Evaluation 2 for the agent using migration to the next interaction partner for each interaction does not change compared to scenario 1 because all communications are realized by local procedure calls. Nevertheless in scenario 2 the execution time of the always migrating agent has become better than the execution time for the RPC-only agent. The execution time for the agent in evaluation 3 with one single migration to location $L_2$ is in scenario 2 not optimal any more. The additional execution time for the increased number of remote procedure calls to location $L_4$ supersedes the amount of time needed for a second migration to location $L_4$ before interaction $I_4$ in order to realize these communications by local procedure calls, as it is done in evaluation 4.

## 5.4 Experimental Validation

To validate the introduced performance model, we have performed measurements of the execution time of mobile agents running on the prototype agent system implementation.

### 5.4.1 Experimental Setup

Since we did not have access to a mobile device with a running mobile agent system, the mobile computing scenario from Section 5.3 was imitated by placing location $L_0$ on a host in the US (ICSI, Berkeley) and a cluster of four locations $L_1$ to $L_4$ on hosts on our local area network at the IPVR. This setup provides a slow and low-bandwidth connection from location $L_0$ to the cluster compared to the fast and high-bandwidth connection inside the cluster. The values for delay δ(x,y), throughput τ(x,y) and the marshalling overhead μ for previous interactions are measured by the mobile agent system and are accessible by the agents for further usage.

Two interaction sequences were defined with similar characteristics as the interaction sequences in Section 5.3. Table 5 lists interaction sequence $S_3$ which is characterized by a larger amount

Table 5: Sequence of interactions $S_3$

| $i$ | $R_i$ | $m_i$ | $B_{req_i}$ | $B_{rep_i}$ | $\sigma_i$ |
|---|---|---|---|---|---|
| 1 | $L_1$ | 1 | 700 | 3000 | 1 |
| 2 | $L_2$ | 1 | 3500 | 15000 | 1 |
| 3 | $L_3$ | 1 | 700 | 3000 | 1 |
| 4 | $L_4$ | 1 | 3500 | 15000 | 1 |
| 5 | $L_0$ | 1 | 3000 | 700 | 1 |

of communication with service agents on location $L_2$ and $L_4$ than with those on locations $L_1$ and $L_3$. The resulting scenario corresponds to scenario 1 of Section 5.3.1. Table 6 lists interaction

Table 6: Sequence of interactions $S_4$

| $i$ | $R_i$ | $m_i$ | $B_{req_i}$ | $B_{rep_i}$ | $\sigma_i$ |
|---|---|---|---|---|---|
| 1 | $L_1$ | 1 | 700 | 3000 | 1 |
| 2 | $L_2$ | 5 | 700 | 3000 | 1 |
| 3 | $L_3$ | 1 | 700 | 3000 | 1 |
| 4 | $L_4$ | 5 | 700 | 3000 | 1 |
| 5 | $L_0$ | 1 | 3000 | 700 | 1 |

sequence $S_4$ where the single large communication with agents on location $L_2$ and $L_4$ is replaced by five smaller communications such that the total amount of bytes transmitted is not changed. This scenario corresponds to scenario 2 of Section 5.3.2.

To undertake the measurements, a static agent is started on location $L_0$. The static agent starts other mobile agents which have to interact with service agents on locations $L_1$ to $L_4$ according to the mobile computing scenario in Figure 5.5. The static agent measures the time from the initialization of a mobile agent until to the delivery of the corresponding reply message by RPC. The characteristics of the mobile agents are $B_{code}$=10kBytes, $B_{data}$=32kBytes, $B_{state}$=0Bytes (due to the restricted migration used). The agent code does never need to be transmitted ($P$=0%) and the size of the agent data does not change (σ=1).

To carry out their mission, the mobile agents followed one of three mobility strategies: The 'RPC-only'-agent remained on location $L_0$ and used remote procedure calls for each interaction. The 'always-migrate'-agent always migrates to the next interaction partner and then uses local procedure calls. The 'optimized'-agent uses the performance model for the execution time $T_{Seq}$ to decide when and to which location it should migrate.

### 5.4.2 Experimental Results

The execution times for interaction sequence $S_3$ are shown in Table 7. The measurements are averaged over 50 runs of the mobile agent for each of the three mobility strategies. Similar to the results obtained for $S_1$, the ‘RPC-only’ strategy is faster than the ‘always-migrate’ strategy and the ‘optimized’ strategy offers only a small improvement. Using the ‘optimized’ strategy the mobile agent migrated exactly once in 49 of the 50 runs. 47 times it migrated to location $L_2$ and 2 times to location $L_4$, an observation which reflects the dynamically changing network load measured by the mobile agent system and used by the ‘optimized’ strategy.

Table 7: Measurements f. interaction sequence $S_3$

| mobility strategy | average time [ms] | standard deviation [ms] |
|---|---|---|
| ‘RPC-only’ | 7501 | 748 |
| ‘always-migrate’ | 9793 | 1140 |
| ‘optimized’ | 7462 | 1341 |

The execution times for the three mobility strategies applied to interaction sequence $S_4$ and averaged over 50 runs of the mobile agent are listed in Table 8. As expected, here the ‘always-migrate’ strategy performs better than the ‘RPC-only’ strategy and the ‘optimized’ strategy offers another improvement. Using the ‘optimized’ strategy, the mobile agent migrated exactly once in 30 cases (11 times to $L_2$, 6 times to $L_3$ and 13 times to $L_4$) and in 20 cases the mobile agent migrated exactly two times (14 times to $L_2$ and $L_4$; 6 times to $L_3$ and $L_4$). Again, the dynamically changing measured values for delay and throughput explain this observation.

Table 8: Measurements f. interaction sequence $S_4$

| mobility strategy | average time [ms] | standard deviation [ms] |
|---|---|---|
| ‘RPC-only’ | 19127 | 1516 |
| ‘always-migrate’ | 11394 | 1414 |
| ‘optimized’ | 10953 | 1341 |

## 5.5 Conclusion

We have introduced a performance model for mobile agent systems where agents can alternatively use remote procedure calls or agent migration for the interaction with partners on different locations. The model was first used to identify situations where a single agent migration has advantages compared to a single remote procedure call. This is basically the case when the amount of data to be processed is large compared to the size of the agent and if the selectivity of the agent, i.e. the ability of the agent to reduce the size of the reply by remote processing, is high. Then the model was extended to describe a sequence of interactions. From this model the conclusion can be drawn that an alternating sequence of remote procedure calls and agent migrations performs better than a pure sequence of remote procedure calls or a sequence of agent migrations. This result was confirmed by measurements on the  mobile agent system prototype implementation. The performance model for the execution time was also used to optimize the mobility behaviour of the mobile agent in a given interaction scenario. The optimization was successful although dynamic fluctuations of measured model parameters like network delay and throughput have weaken this result. The performance model could thus be a building block for the optimization of the mobility behaviour of mobile agents.

# 6 Efficient Integration of Services in Mobile Agent Systems

## 6.1 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 service by client agents.

Mobile Agent Systems basically consist of a number of locations and agents (see Figure 6.1.). 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.

Agents are active, autonomous software objects, that reside (and are processed) on locations. They can communicate with other agents either locally inside one location ore 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. In this section, these mechanisms are provided by the location manager LM, which offers a programming interface for the agent programmer.

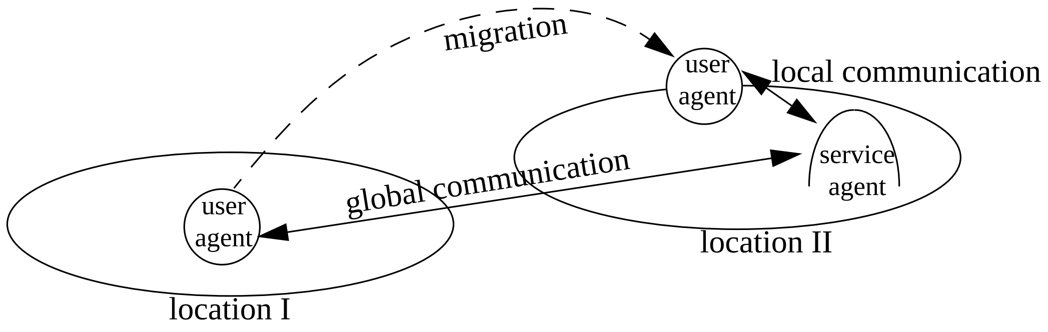
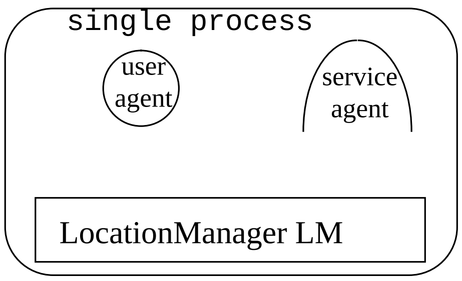


**Figure 6.1.** Mobile Agent System

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.

Service Agents either implement a service (i.e. within the Mobile Agent System) or they are the interface to (possibly already existing) services outside the Mobile Agent System, or any combination thereof.

Concerning the discussion about the efficient integration of services, two basic implementation types of locations have to be considered: The first implementation type manages the whole location within a single process (see Figure 6.2). Components of the location, i.e. the location manager as well as the user and service agents residing on this location, are handled by lightweight processes (threads) and the interaction between these components is realized using the shared resources of this single process. Examples of Mobile Agent Systems using this type of single-process location are Aglets[IBM96], Odyssey[GM97], Ara[Pei97] and Mole[SBH97] The second implementation type manages each component of the location within its own process (see Figure 6.3). Interaction between the components must be realized using resources provided by the underlying operating system like messaging or (distributed) shared memory. An example of a Mobile Agent System using this type of multiple process locations is ff-Main[LDD95].



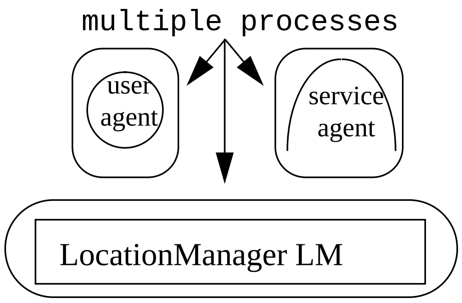


**Figure 6.2** Single-Process Location **Figure 6.3** Multiple-Process Location

This report addresses the efficient integration of existing services and the efficient implementation of new services into Mobile Agent Systems. The following issues are considered: The parallel processing of service requests is discussed in Section 2. In Section 3 service agents are considered as interface between different language domains. Section 4 discusses the efficient implementation of context free versus context sensitive services. Sections 5 reviews authentication/authorisation issues.

## 6.2 Parallel Processing of Service Requests

Requests to the service agent should be processed as fast as possible. While the client agent communicates with a single, unique service agent, this service agent has to deal concurrently with a large number of requests from different client agents. The incoming requests to the service agent therefore have to be distributed among the available execution units, i.e. the processes and/or threads that execute the service agent.

### 6.2.1 Single Execution Unit

In this simple service architecture, the service agent executes requests serially:

```
WHILE (TRUE)
   Wait For Request
   Do Work
   Send Reply
END
```

If only a single execution unit (thread or process) is available for the execution of the service agent, the behaviour of service agents in single-process locations and multiple-process locations is equivalent: Either the service agent is ready to serve or it is busy. If the service agent is ready

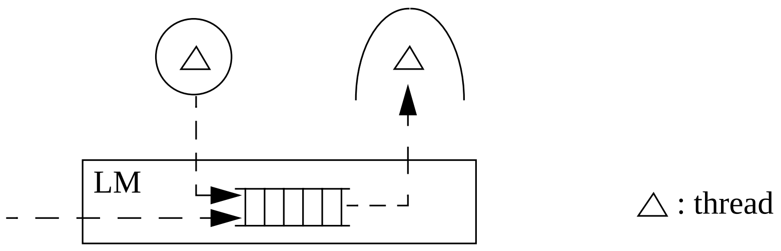


**Figure 6.4** Serial execution of service requests

to serve, the request can be processed immediately while the status of the service agent is switched to busy. While the service agent is busy, incoming service requests cannot be processed and must either be denied (or discarded) by the location manager, or the location manager must somehow buffer the request until the service agent is ready again. The latter is normally implemented using a queue where the location manager can store requests and from which the service agent can poll requests as soon as it is ready to serve. Again, the location manager has to deny (or discard) service requests, if the queue is full. The implementation of a service using queues can deal with temporary peaks for computational cheap or seldom requested services, but it does not scale well with the number of requests.

A special case is given if the service agent interfaces an (existing) multithreaded server. In this case the service agent can directly forward incoming requests to the server and forward replies to the correct client agent.

```
WHILE (TRUE)
    ServiceRequest? -> forward to server
    Service Reply? -> forward to correct client agent
END
```

### 6.2.2 Multiple Execution Units

Incoming requests are processed in parallel. The execution units can either be light-weighted processes (threads) inside a single address space or full processes with separated process spaces [OHE96]. The processing by threads has the advantage that threads are cheap regarding creation of threads and context switching between threads. Additionally, the access to common data is simple and also cheap. At first sight, the parallel processing of requests by several processes seems to have only disadvantages. The usage of common data in full processes needs the usage of (expensive distributed) shared memory on the inter-process level (not always available) or the usage of external mechanisms like e.g. data bases. The creation of processes and the context switches between processes are expensive compared to the same operations on threads. But, on the other side, the usage of full processes to process incoming requests in parallel offers some advantages. If a process processing a request fails, it fails independently, that is, no other process of the service agent is affected [Ber97]. Furthermore, the usage of processes may be preferred either if the implementation of threads is inefficient or to allow the efficient distribution of the processes onto loosely coupled multi processor like e.g. the Tandem Himalaya.

The assignment of a request to an execution unit is performed by a scheduler. The scheduler can be placed either in the location manager or in the service agent (see Figure 6.5). If the scheduler is placed in the location manager, it has to be only implemented once, but it has to manage all execution units of all service agents. If the scheduler is placed in the agent, the location manager forwards incoming requests for the service agent to the agents scheduler. This allows individual scheduling policies of the agent and additionally gives some flexibility in case of using native code for the service implementation in multi-threaded service agents (see section 3.2). The disadvantage of placing the scheduler into the service agent is, that every service agent has to implement its own scheduler (or use a standard scheduler). Additionally, it does not perform very well in case of a multiple process realization of a service agent in a multi-process location, because requests have to be forwarded from the location manager to the process serving the request via the scheduler of the agent (also realized in a process).

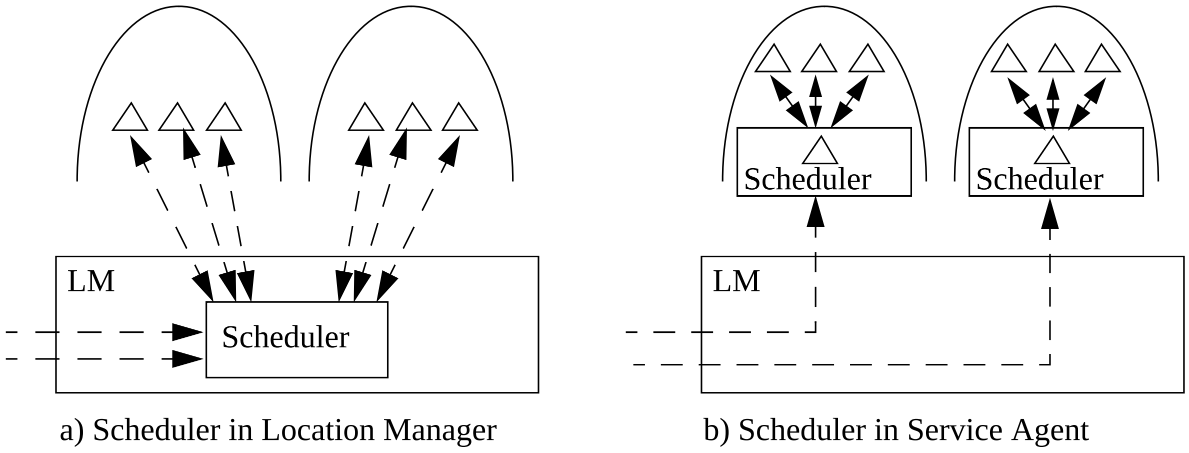

LM
Scheduler
a) Scheduler in Location Manager
Scheduler
Scheduler
LM
b) Scheduler in Service Agent

**Figure 6.5** Assignment of requests by a scheduler

Let us now consider a single-process location, i.e. where most of the agents of one location are assigned to the same process. In Figure 6.6 the multiple execution points of the service agent are realized by threads. Please note that a service request from an agent residing on the same location would bring its own thread to the service agent (solid arrow) while the service request from another location would need to be assigned to a thread of the service agent by the scheduler (dashed arrow), which resides in the location manager in this example. If the multiple execution points of the service agents are realized by separate processes (see Figure 6.7), requests to the (dashed) service agent are forwarded transparently by the scheduler to the service agent processes. Each service agent process must contain next to the service agent also a service manager (SM). The service manager is a reduced version of the location manager and handles the communication with the location manager. The service manager provides the same API for the service agent as the location manager.

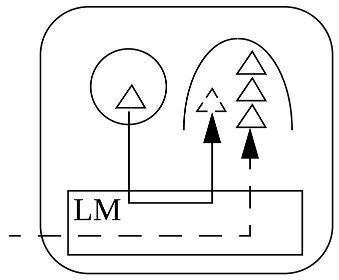


**Figure 6.6** Multi-threaded service agent in single-process location

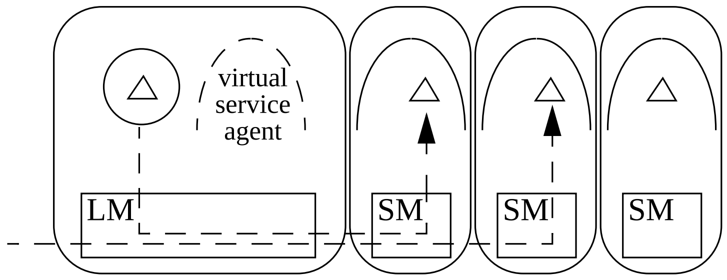


**Figure 6.7** Virtual service agent realized by multiple processes in single-process location

Let us now consider a multiple-process location, where all agents and the location manager (containing the scheduler) are assigned to different processes. In Figure 6.8 the multiple execution units are realized by threads and in Figure 6.9 the execution units are realized by processes. In both cases the scheduler has to keep track of the number of available threads/processes and has to return replies to requests to the correct client agent. Please note that the service agent processes do not need an additional service manager since communication with the location manager in multiple-process locations is realized by inter-process communication anyway.

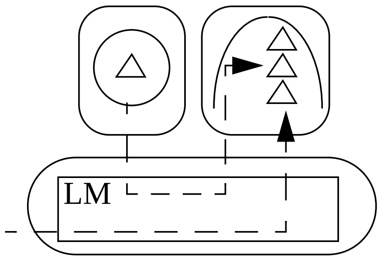


**Figure 6.8** Multi-threaded service agent in multiple-process location

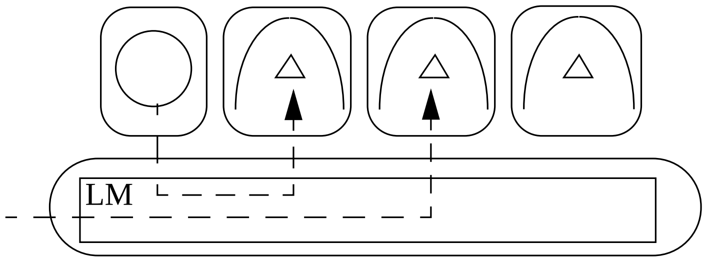


**Figure 6.9** A service agent realized by multiple processes in multiple-process location

The efficiency of the different architectures depends on the architecture of the machine running the location. The architecture in Figure 6.6 is likely to be the right choice for single processor machines or tightly coupled multi processor machines, because no process context switches are needed while processing requests from different clients. The architecture Figure 6.7 on the other side is very convenient for the distribution of the execution units on the processors of loosely coupled multi processor machines or for the distribution of the execution units on different (local) machines. The drawbacks are the same in Figure 6.8 and Figure 6.9, but there are additional process context switches necessary for communication between agents and the location manager resulting in poor performance.

### 6.2.3 Multiple Execution Units with Binding

The assignment of execution units to a client agent by the Mobile Agent System can either be done dynamically for each request or statically for a sequence of requests. With dynamic assignment each request of the same client agent could be processed by a different execution unit. With static assignment, successive requests of a client agent to a service agent are processed by the same execution unit of the service agent.

To support this at the interaction level, a possibility to establish a binding between client agent and service agent, providing some sort of communication association, can be offered. If a client agent wants to communicate to a service agent, it may invoke a binding by

```
handle = Bind(ServiceAgent)
do several requests
UnBind(handle)
```

If no binding exists between client agent and service agent, always dynamic assignment from client agent to execution unit of service agent is used. If a binding is established, the use of static or dynamic assignment of the execution units depends on the service agent. If the service agent interfaces a service which preforms its own scheduling, no static assignment is possible. But if the scheduling of requests is performed by the service agent (or the location manager), the execution units can be assigned statically to the client agent. In addition to the static assignment of execution units, the communication associations can be used to manage related data (see session concept in Section 6.4.2 ).

Figure 6.10a. shows the binding procedure of a remote client agent to a service agent in a single-process location. The client agent on Location 1 sends a Bind request to the (virtual) service agent on Location 2. The location manager returns a handle which is used (see Figure 6.10b.) in subsequent requests to access the service agent process via its service manager (if processes are used).

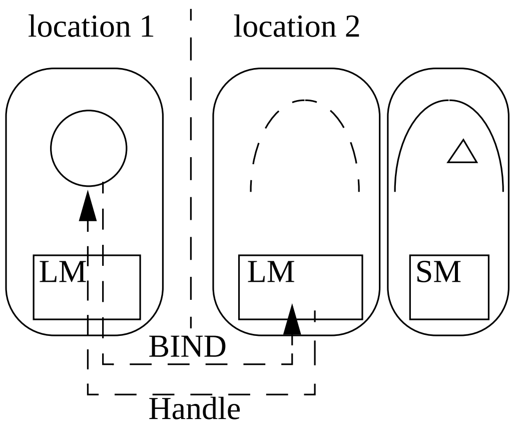


**Figure 6.10a.** Binding in single-process location

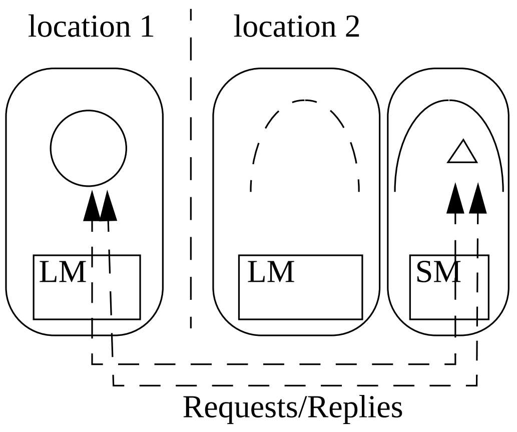


**Figure 6.10b.** Subsequent requests

The same binding procedure is possible for multiple-process locations. While for the binding procedure both location managers are involved (see Figure 6.11a.) in subsequent requests the service agent is accessed directly by inter-process communication.

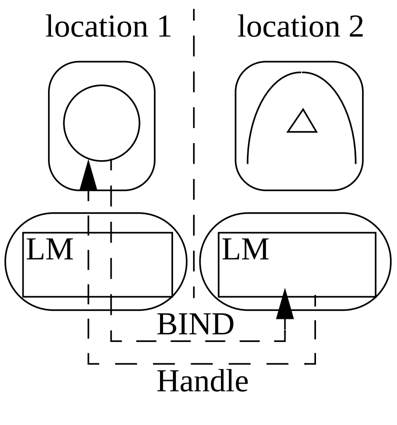


**Figure 6.11a.** Binding in multiple-process location

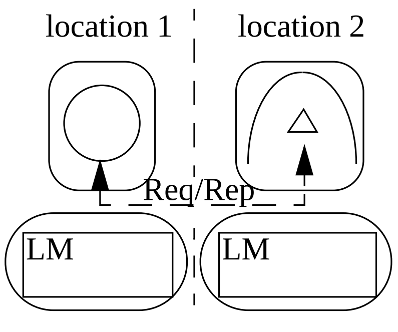


**Figure 6.11b.** Subsequent requests

The use of binding hasn't too much impact on performance in service agents using multiple threads to execute the client requests. For service agents using multiple processes for the request execution, the binding offers the possibility that the client agent directly contacts the assigned process avoiding the overhead needed by the forwarding of the request via the location manager.

### 6.2.4 Multiple Execution Units and Server Classes

The execution units for the service agents can either be taken from a static pool of execution units (called a server class if a pool of processes run the same program to service the requests [Ber97][Tan96a]) or they can be created dynamically upon request.

If a static pool of execution units is used, a fixed number of them has to be created at the start-up of the service agent. If all execution units are assigned to a request, further incoming requests have to be queued until a execution unit is free again. This has the advantage that overhead for the creation and killing of threads/processes is avoided, but this method does not scale well in case of request peaks, in particular in the case of a static binding where execution units might be busy with a client agent for longer time intervals.

If an execution unit is created for each request or binding, each request can be processed immediately, but overhead is introduced for thread/process creation.

A mixed strategy goes as follows: a static pool with $n$ execution units is created at start-up of the service agent. If necessary (e.g. if the queue always contains at least $m$ requests) additional execution units can be created dynamically. If the load is reduced (e.g. if there are always at least $m$ execution units idle) the additional execution units can be killed until only the initial $n$ execution units remain in the pool.

## 6.3 Embedding of Services into Interpreted Agent System

Mobile agents are generally written in an interpreted language like Java or Telescript. The reasons therefore are the ability to enable agents to migrate between locations on different system architectures (which is difficult to realize for native code agents) on one side and the fact, that the safe execution of code is much more easy to realize in interpreted languages on the other side. But the usage of an interpreted language for service agents is not very efficient since interpreted languages are generally slower than compiled languages, and it is not possible for existing services which might only be available or accessible in native form(e.g. subroutine libraries). Thus next to the implementation of service agents in the interpreted language of the Mobile Agent System their implementation in a language which will be compiled into native code or in a mixed form with interface between interpreted and native part of the service agent should be provided.

### 6.3.1 Embedding of Interpreted Code

This allows the implementation of hardware independent service agents, which thus allows the installation of a service on arbitrary locations of the Mobile Agent System. It can be applied for value added services or services that do not need a special infrastructure. The efficiency of such services can only be increased by parallelization.

### 6.3.2 Embedding of Native Code by Interpreter Interface

This approach is platform dependent and can only be realized for dedicated environments. Two methods for the interface between interpreted language and the native code have to be distinguished. Either the interface between interpreted language and native code is provided by the interpreted language (i.e. the interpreter allows the call of native procedures/methods, see Figure 6.12) or native code is executed in a separate process and the interface is realized by inter-process communication (see Figure 6.13). The latter case is not discussed further since it is similar to a pure interpreted service agent that communicates with another service.:

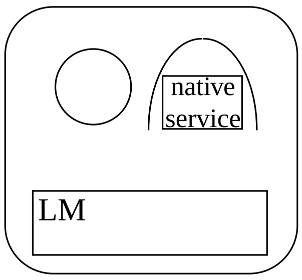


**Figure 6.12** Call of native service procedure

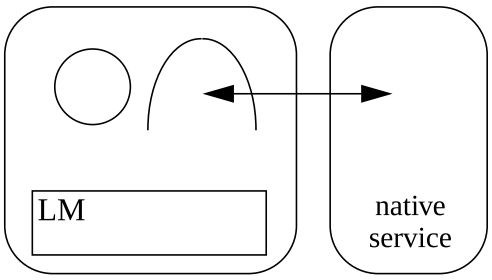


**Figure 6.13** Call of native service process

This mixed language programming approach can be used for the interface to existing services as well as for the efficient implementation of new services. The interface to existing services can be done either by contacting the service using existing native code libraries (e.g. contacting a TP-monitor) or by calling the procedure that implements the service directly. The efficient implementation of new services can vary from implementing the service mostly in interpreted code, providing only small but computation intensive parts in native procedures to the complete implementation of the service in native code which is called from small interpreted envelope procedure.

An interesting question arises when using an own scheduler in a multi-threaded service agent: Should the assignment of requests to threads be handled in the interpreted part or in the native code part of the service agent? If the implementation of the service consists of interpreted code with embedded calls to native code, the scheduler has to be implemented in the interpreted language (see Figure 6.14). If the service is completely realized in native code, it is possible to realize the scheduler in native code being called from small interpreted envelope procedure (see Figure 6.15). Please note that in single-process locations, the scheduler has to take care for requests issued from client agents on the same location bringing their own thread.

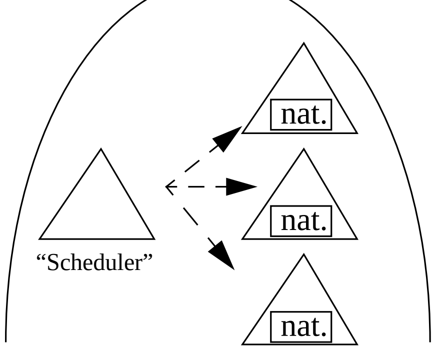


**Figure 6.14** Interpreted scheduler

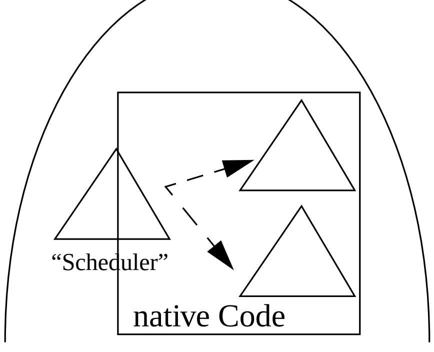


**Figure 6.15** Mixed code scheduler

### 6.3.3 Embedding of Compiled Code by Process Call

In contrast to the previous section, where a service agent that was implemented completely by native code is nevertheless called from an interpreted envelope procedure and thus by the corresponding interpreter running in the process, there exist cases where the indirect call through the interpreted envelope procedure is not necessary and the native code can be accessed directly.

This is in particular the case for the parallel implementation of services by multiple processes. In single-process locations, the native code service agent must additionally implement a pseudo service manager as interface to the remaining location (see Figure 6.16) In multi-process locations, the native code service agent must be able to communicate using the inter-process protocol of the Mobile Agent System.:

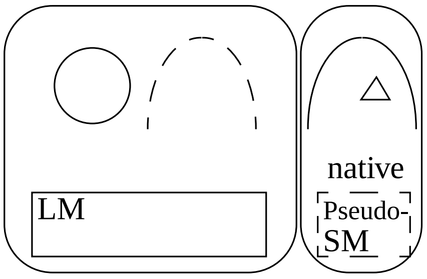


**Figure 6.16** Native code agents in single-process locations

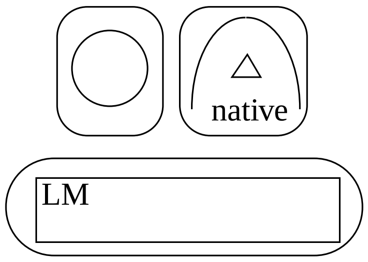


**Figure 6.17** Native code agents in multiple-process locations

A further possibility to integrate native code services is the implementation of a whole `simulated' location in native code (see Figure 6.18). To the outside world a simulated location behaves like a normal location, except that it is not possible for agents to migrate to a simulated location. A simulated location is only populated by a set of (virtual) service agents, which can only be accessed through a pseudo location manager by agents residing on other 'real' locations (see Figure 6.18). The pseudo location manager of a simulated location must provide to other locations the same interface as the location manager of a real location. The pseudo location manager handles incoming requests to the virtual service agents by redirecting them to corresponding compiled service procedures.

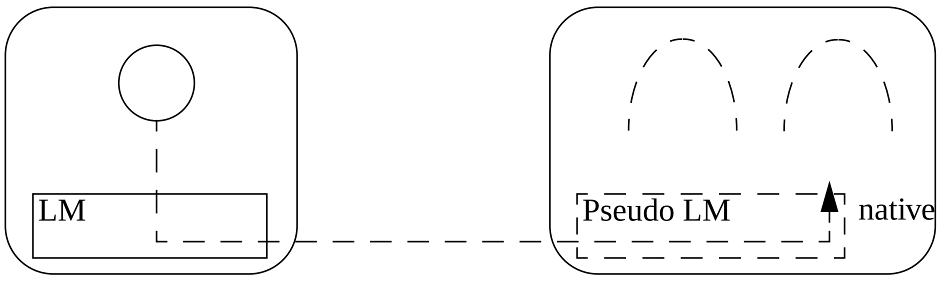


**Figure 6.18** Access to a simulated location

## 6.4 Context-free and Context-sensitive Services

Services can either be context-free or context-sensitive. Successive requests to context-free services are independent from each other (e.g. a service that returns the time). For context-sensitive services, successive requests are dependent from each other. For example a file writing service: If a client writes into a file by successive requests, somewhere the file pointer (i.e. the position where the next information has to be stored) must be maintained. If this information is resubmitted with each request (that means the context is managed by the client), the file writing service is context-free. (e.g. NFS [RFC1094]). If the information is maintained by the service itself, the service is context-sensitive (RFS in UNIX System V).

Concerning the integration of services into Mobile Agent Systems, context-free services can be implemented straightforward, but context-sensitive services introduce some interesting problems: Where should the context be stored, how can it be accessed, and how long must it be kept? The following discussion of context sensitive services is based on an agent system with single-process locations. Analogous considerations hold for multi-process locations.

### 6.4.1 Where should the context be stored?

In *single execution point service agents*, the context data of client agents is stored in a data structure containing the contexts of all client agents. This data structure is stored in the memory area belonging to the process executing the service agent. For each request, the client agent name is mapped to the appropriate context data used to process the request. In *multi-threaded service agents*, the context data of all client agents is also stored in the memory area belonging to the process executing the threads of the service agent (see Figure 6.19).

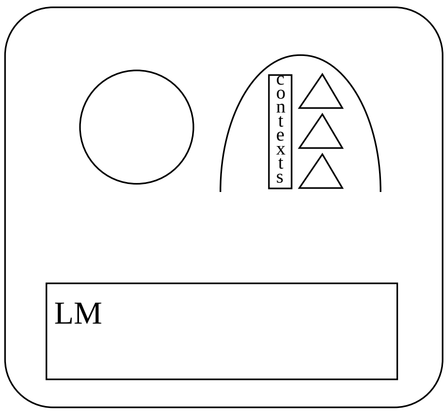


**Figure 6.19** Context in one process

*Multi-process service agents* with access to (distributed) shared memory can also store the context data of their client agents in the shared memory accessible to all service threads (see Figure 6.20). It has to be noted that the access to distributed shared memory can be very slow[Tan95].

If *multi-process service agents* have no access to (distributed) shared memory (or the access to the shared memory is too slow), it has to be distinguished if the binding of the client agents to the process is dynamic or static (see section 2.3). In the case of dynamic binding, the location manager has to coordinate client and its context. Therefore, the context has to be submitted from the location manager to the service process for each request (see Figure 6.21) and, after the execution of the request, the context (or the changes to the context) has to be stored back to the location manager. Depending on the size of the context information, this introduces some overhead for each request. In the case of static binding of a client to a process, the context can be established during the binding in the allocated process (see Figure 6.22).

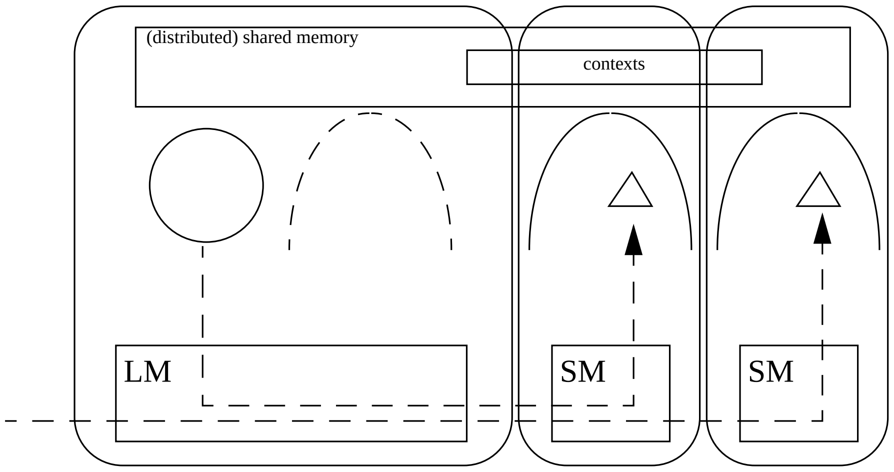


**Figure 6.20** Context in shared memory

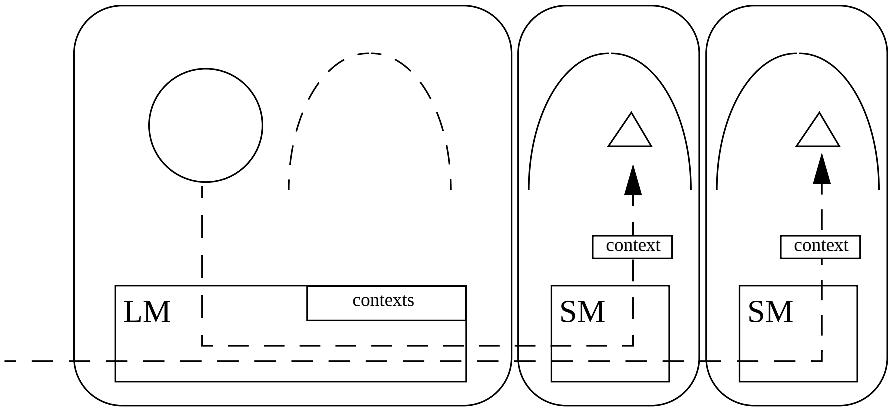


**Figure 6.21** Context managed by LM

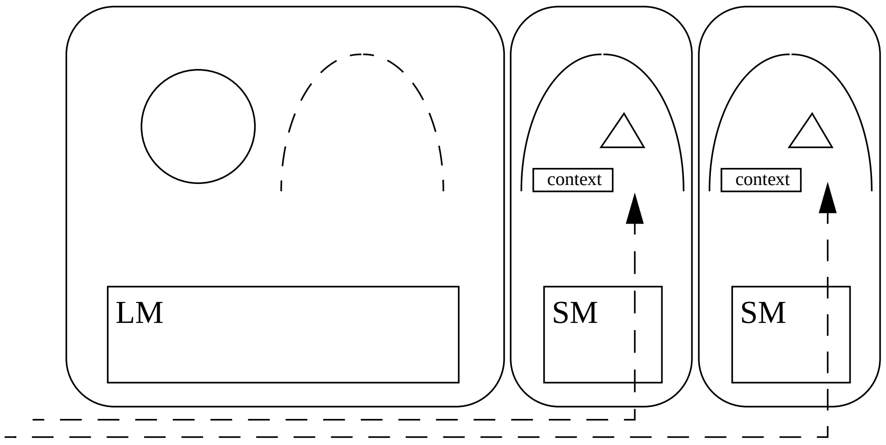


**Figure 6.22** Context in service process

### 6.4.2 How long should the context be kept?

The question when to release the mapping between client and client specific context is simple in case of the static binding of client and service agents: release with ending of the binding. In the other cases additional concepts have to be introduced. In this context the *session-concept* [Mul90] might be useful. The client agent opens a session with the service agent, communicates with the service agent and then ends the session [BH+97]. The session can either bind the context of a context-sensitive service, the execution unit or other information like e.g. authentication/authorization (see Section 6.5 ). Inside a session next to the client/server interaction pattern other communication patterns via messages (peer to peer communication) could be realized (see e.g. context sensitive services in the Tuxedo-environment[Tan96a])

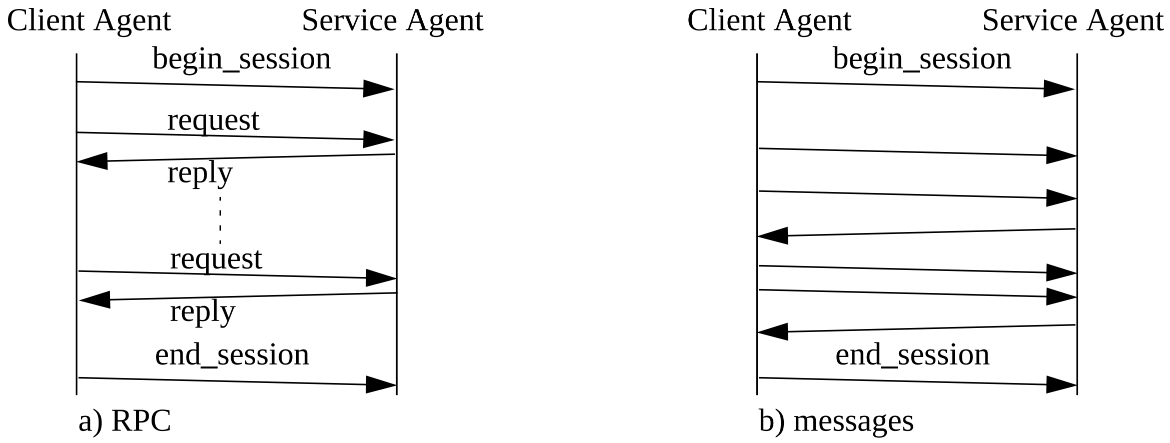


**Figure 6.23** Different interaction types in sessions

## 6.5 Authentication/Authorization

Important aspects in distributed systems and in particular in Mobile Agent Systems are the authentication (who am I) and the authorisation (what am I allowed to do).

In Mobile Agent Systems the following approaches have to be considered:

- The rights of a client to access the service agent are controlled by the service agent (using some existing security services). The service agent provides the access rights for the underlying services. In this scenario, the service agent is responsible for the correct access protection of the underlying service.

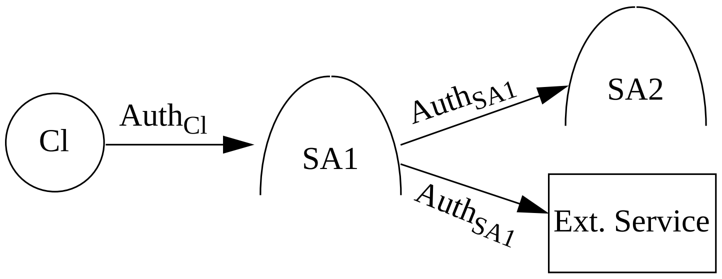


**Figure 6.24** Authorization by Service Agent

- The service agent uses the security attributes (identity attributes, permissions,...) of the client to access the underlying services. Therefore, the service agent needs no access rights for the underlying services

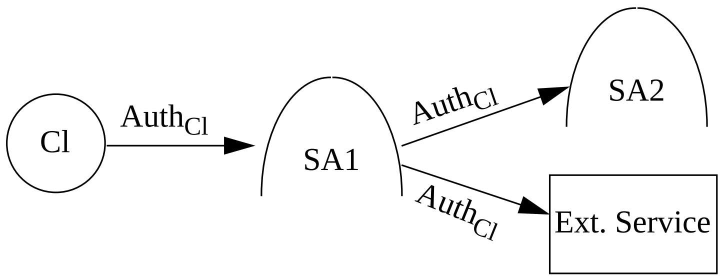


**Figure 6.25** Authorization by underlying service

- A combination of the other approaches: The client provides some security attributes. These may be used by the service agent e.g. to control the access rights of the client or to bill the client for the service. To access underlying services, the service agent uses a combination of its own security attributes and the security attributes of the client.

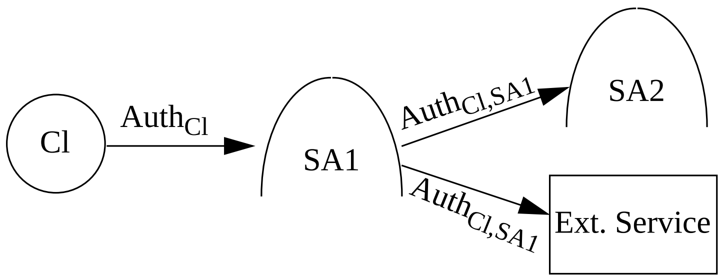


**Figure 6.26** Authorization by Service Agent and underlying service

Several security services are available [Sch96,Tan96b]. The security service used by existing services depends in general from the organisation which provides the service. Therefore, a generic mechanism analogous to the CORBA security services [OMG97]is proposed:

The client authenticates itself using some kind of authentication mechanism. As a result, it receives a credential, containing security attributes of arbitrary form (access right, identity attributes,...). This credential is passed automatically to the service agent during communication. The service agent can use the credential e.g. to authenticate the client or to check its access rights. The service agent calls subsequent services using a credential containing some security attributes of its own and some security attributes of the client (see Figure 6.27).

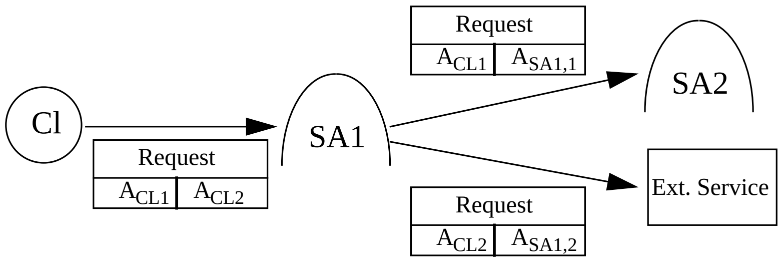


**Figure 6.27** Authentication/Authorization using credentials

The credential may be passed to the service agent (and verified by the service agent) either at each request or, if sessions or static bindings of execution units are used, only at the establishment of the association. In the last case, the service agent stores the credential until the association is terminated.

## 6.6 Replicated Locations

The architectures presented so far are centralized (at least, there exists only one location manager per location) and therefore don't scale very well. A solution offering enhanced scalability are replicated locations. A replicated location (or logical location, see Figure 6.28) consists of several parts, also called sublocations. These sublocations are executed on different processors (loosely coupled multi processor machines, e.g. Tandem Himalaya; multiple workstations with high speed connection; ...).

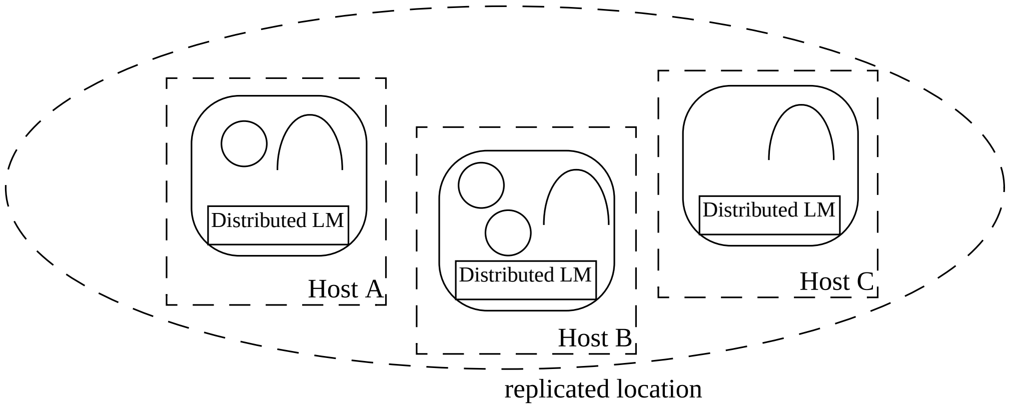


**Figure 6.28** Replicated Location

In contrast to data replication, where each replica contains the complete data of the replicated logical object, each part of a replicated location (called sublocation) contains only some of the agents residing at the location (each agent resides at exactly one sublocation). However, the replication is transparent for the agents, i.e. the communication to and between agents residing at the location and the migration to the location are handled transparently.

The management of the replicated location is handled by the location managers residing at each sublocation, which realize a distributed location manager for the (logical) location. To contact the replicated location, any of the sublocations may be contacted. If, for example, an agent wants to migrate to the replicated location, the source location manager contacts the location manager of any of the sublocations. Depending on the load of the sublocations, the migrating agent is then placed on an appropriate sublocation (see Figure 6.29).

Communication between agents (see Figure 6.30) on the replicated location which reside on different sublocations is handled by the location managers of the communicating agents. If an agent from a remote location wants to communicate with an agent on a replicated location and contacts the wrong sublocation (not containing the communication partner), the communication can be either forwarded to the correct sublocation or the remote location may be informed about the appropriate sublocation. While the forwarding of the communication is preferable if only one message (or a pair of messages) is exchanged between the communication partners, the redirection to the correct sublocation is the appropriate method if several messages are exchanged.

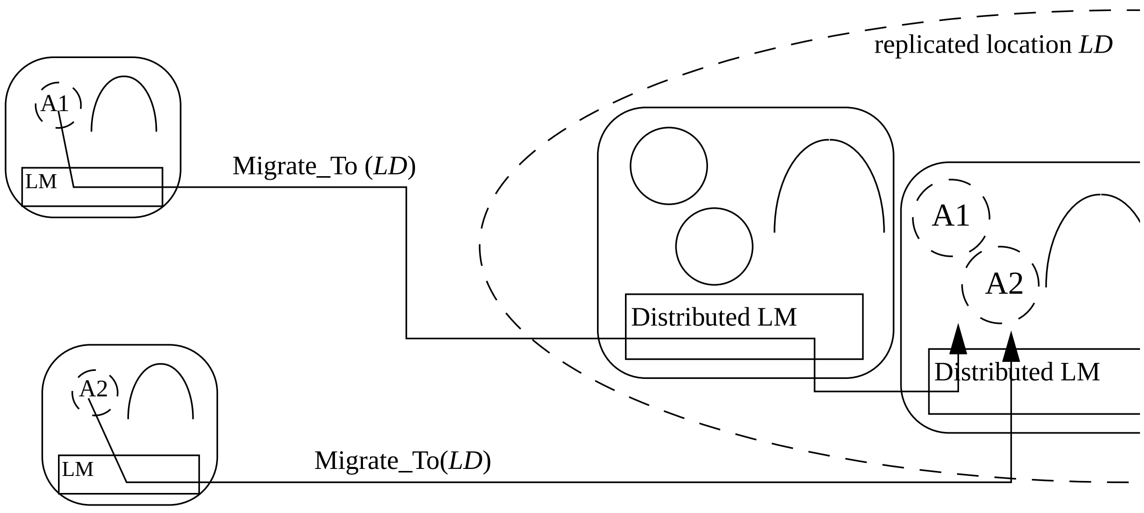


**Figure 6.29** Migration onto the replicated location *LD*

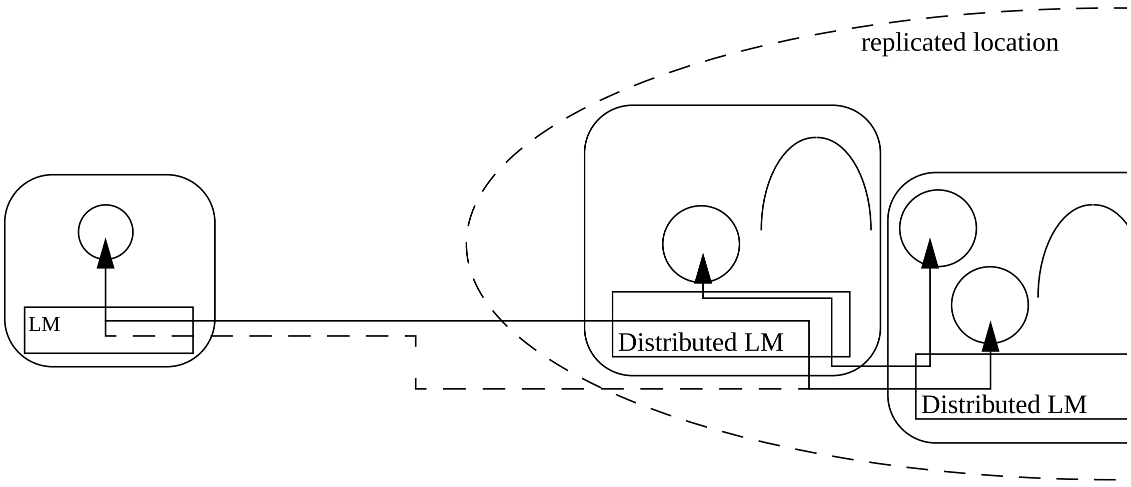


**Figure 6.30** Agent communication between replicated locations

Figure 6.31 shows three alternative structures for replicating a location offering three services. While the first structure is appropriate for easy-to-replicate services, the other structures suit for services which cannot be replicated or for interfacing already existing services.

The architecture of the replicated locations offers several advantages. First, it improves the scalability of a location, enabling much more agents to migrate to this location and requesting the services of this location (i.e. improving the efficiency of the offered services by improving the efficiency of the location). Second, the availability of the offered services increases because the service is available as long as at least one sublocation is available (only if the service implementations are distributed).

Two obvious disadvantages of the replication are the more complex implementation of the locations and the expensive “local” communication between agents on different sublocations of one logical location. Another disadvantage of the replication is a somewhat “strange” error semantic. If only one sublocation of a distributed location fails, only the agents residing at this

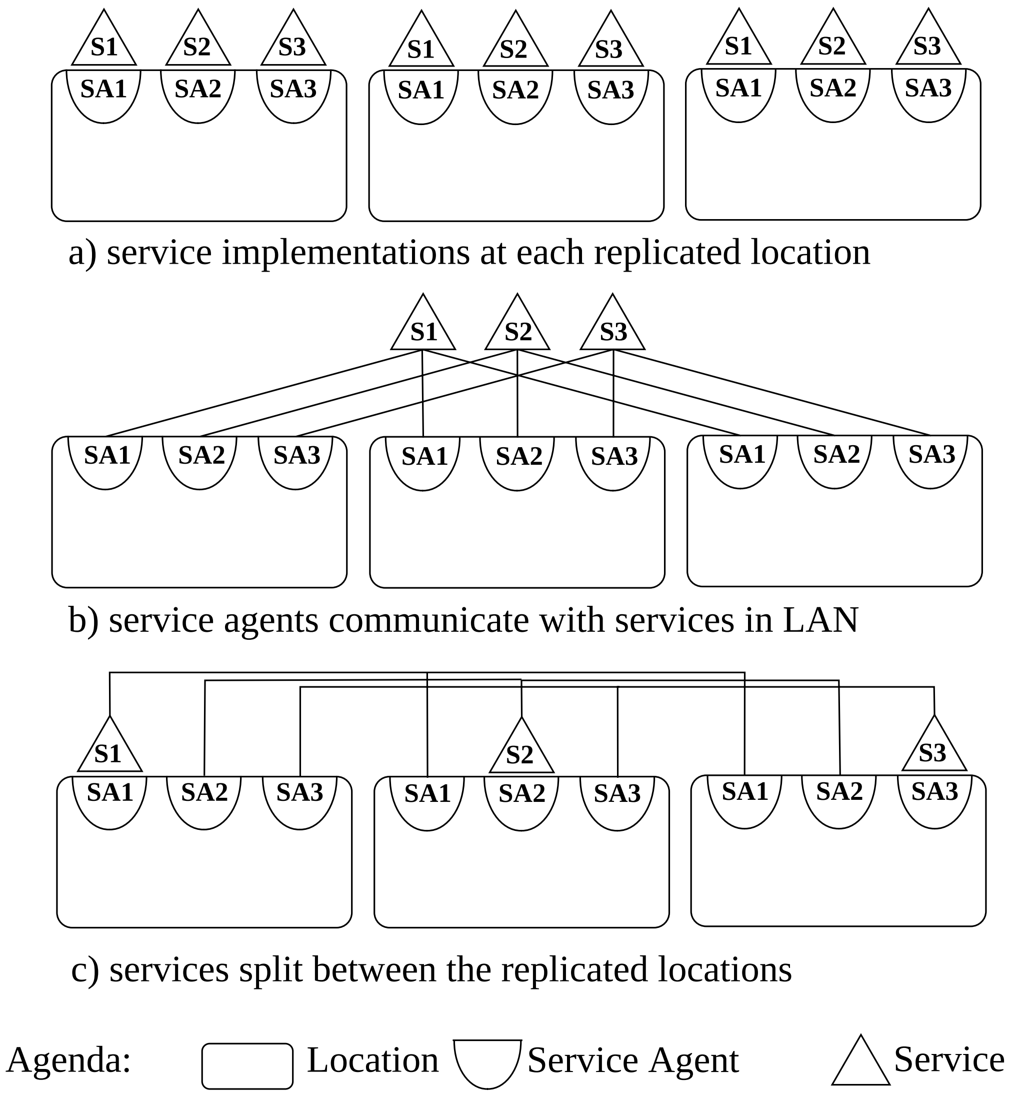

S1
S2
S3
SA1
SA2
SA3
a) service implementations at each replicated location
b) service agents communicate with services in LAN
c) services split between the replicated locations
Agenda:
Location
Service Agent
Service

**Figure 6.31** Alternative structures for replicating a location

sublocation are unreachable. In this case, instead of being transparent for the agents, effects of the replication are visible to the agents.

## 6.7 Conclusion and Work in Progress

In this section, several possibilities for the efficient integration of services were presented. While some approaches seem to be appropriate only for small classes of services (e.g. service agents with single execution unit for simple services), other approaches perform different on various system architectures or may be only implemented some system architectures.

These different properties of the presented architectures are the subject of the current work in progress. Hereby, most of the presented architectures will be realized using some sample services. The performance of the implementations will be measured and evaluated on different system architectures.

# 7 Security

Mobile agent systems are expected to become the base platform for an electronic services framework, 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 fullfilled even in the traditional, non-electronic commerce, but the anonymity of a worldwide communication network and the ease of the automatic exploitation of security gaps in electronic applications make it necessary to meet this demand when it comes to commercial transactions done by computers. Therefore, we investigated the security aspect of Mobile Agents for critical problems that may prevent a usage of this technology even if we would have a mature middleware for e.g. Electronic Commerce applications.

Mobile agent systems are platforms that allow mobile agents, to migrate between different nodes of the agent system. On a node, a mobile agent can interact with other agents locally (and globally in some systems) or it can migrate to another node. Interaction consists in communicating with other agents, requesting or providing services or creating new agents or terminating existing ones. Nodes, or mobile agent hosts are the “building block” of the agent system; each host can be maintained by another institution.

When it comes to security, security in mobile agent systems can be divided in different security “areas”, e.g. by identifying the different attack “fronts”. These attack fronts distinguish the “parties” involved (e.g. the agent and the host) and allow to identify the possible attacks between the parties. In this report, we distinguish four main security areas (Figure 7.1.):

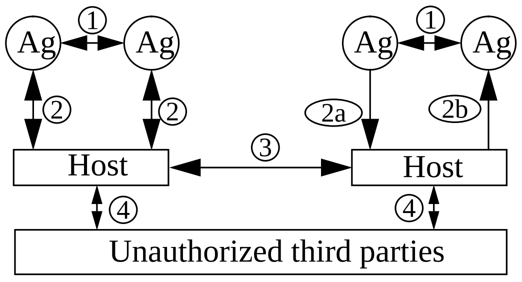


**Figure 7.1.** Security areas of mobile agent systems

1. Security between two agents
2. Security between agents and hosts
3. Security between hosts
4. Security between hosts and unauthorized third parties

In the following, every security area will be characterized by mechanisms, that have to be offered in this area and by techniques, that may implement the mechanisms.

## 7.1 Security between two agents

The problem addressed here is the security of agents against other, malicious agents. Attacks of those may include code and data manipulation by having physical access to the code and data areas of the attacked agent, masking of agents (i.e. faking a wrong identity), cheating (e.g. using a service without paying for it) and denial-of-service (e.g. by filling a message buffer).

As this area does not address new security issues compared to requirements imposed in “traditional” distributed systems, mechanisms that prevent such attacks already exist, e.g.:

- the use of an agent language, that does not allow others to have physical access to “private” data, such as Java or the use of isolated address spaces
- authentication of agents, e.g. by using digital signatures
- employment of a service contract mechanism (e.g. [SL95])

## 7.2 Security between agents and hosts

This area can be divided in two subareas, since the relationships between agents and hosts is not symmetric: Agents are basically programs that are executed by the hosts, the hosts consists of agent code interpreters or at least runtime environments.

### 7.2.1 Security of hosts against malicious agents

The attacks of malicious agents are basically the same as those against other agents (physical manipulation, masking, cheating, denial-of-service). The difference is the greater impact of hosts, since they control the execution of agents. This enables host to stop suspicious agents at any time.

The mechanisms against attacks of malicious hosts are:

- the usage of “secure” languages or isolated address spaces
- authentication by using digital signatures or other cryptographic techniques like the one described for the mobile agent system AgentTcl ([Gra96])
- the usage of accounting and contract mechanisms
- the usage of resource control mechanisms such as limited resource “accounts” and runtime restrictions

The second mechanism embodies a problem compared to “traditional” authentication: As a mobile agent system as a whole may be too large, it cannot be assured that there may not exist two or more agents using the same identification. This security gap can be exploited either by the agent owner to create copies of an existing agent or by attackers (e.g. hosts) that are able to “read” existing agents, such as hosts.

### 7.2.2 Security of agents against malicious hosts

In this security area, the attacker is the host itself. The host can observe every step the agent takes, read every bit of code, data and state, and even manipulate the way the agent works, as it, among other things, interprets the agent code. Attacks include “normal” threats like masking and cheating, but also privacy breaking (the host may read secret keys or electronic money), code, data and state manipulation (e.g. taking away electronic money or implanting virus code, and executing the agent in a way other than specified by the language. Denial-of-service attacks are possible, too, but it is clear that a host can easily deny the agent to work by simply not to execute it.

The existing security mechanisms are not easily applicable here since the aspect of securing a program against a malicious interpreter is a rather new aspect, and indeed, this problem has rarely occured in applications so far since a program either was invoked by the same authority that maintained the computer environment or because the maintaining institution was trustworthy. Therefore, barely any solution exist in this security area.

*On the other hand, exactly this aspect, the protection of the agent, is a very important one. If we cannot guarantee a user, that secrets, authority and money of its agent is protected, it will not employ this technology, which results in an unattractive system from the service provider's point of view.*

Although currently no solution exists, our investigation of this problem in the context of another research project resulted lately in a very promising approach (see [Hoh97]).

## 7.3 Security between hosts

Like in Section 7.1, possible attacks between hosts include masking, cheating and denial-of-service. However, since hosts are stationary entities, the common mechanisms for authentication, accounting and interaction control can be applied here without modification.

## 7.4 Security between hosts and unauthorized third parties

We have to consider in this area the security of communication between two hosts over an insecure network. Other aspects like masking are included in the areas above. The attacks and mechanisms here are well-known and do not pose new questions. Therefore, this area is omitted in this paper. Overviews and detailed informations can be found in a couple of books, e.g. [Sta95], [Sch96] and [Che97].

## 7.5 Conclusion

Mobile agents are an area where existing technology can be employed, although most of the mechanisms have still to be tailored for that purpose. The only new aspect, the protection of agents from a malicious host, cannot be solved using existing technology. Still, exactly this aspect can prevent the usage of the mobile agents technology in Electronic Commerce applications. Therefore, we intend to develop an approach to solve this problem and our first results are promising.

## 7.6 Future work

For the second year, the investigation of existing mechanisms and the implementation of mechanisms for the other security areas, such as authentication of agents and hosts and resource control for agent execution are planned.

# 8 Future Work

In the second phase, the first phase work packages not yet complete will be finished. Therefore, a complete redesign of the prototype communication subsystem will be realized and, as soon as available, standard Java OLTP mechanisms will be integrated in the agent system. Furthermore, the work packages of the second phase will be realized.

WP 2.1 deals with the design and implementation of agent migration functionality. The currently implemented weak migration functionality already provides the same computational possibilities as a strong migration mechanism. In contrast to the weak migration, where only code and data of the agent is migrated (and therefore the agent is newly started at each location), strong migration offers a transparent migration in the sense that the computation of the agent continues at the destination location behind the migrate command. This kind of migration seems to be easier to use, but the realization of multi-threaded agents (agents with several execution points) is problematic in this case. In this work package, we have to investigate, if strong migration functionality is necessary or if the already implemented weak migration functionality is sufficient for easy and efficient agent development.

In WP 2.2, the already started activity in the security area will be continued.

One of the most important and interesting work in the second year will be the extension of the existing system to provide recoverable agents in WP 2.3. Here, we plan to realize a scalable reliable agent server for Java agents making use of the Tuxedo's server class infrastructure. The server enhances Tuxedo allowing Java agents (consisting of code, data and execution state) or only Java code to be uploaded to the server and to be executed in a secure environment. The reliability of the server is achieved by using reliable, transactional mechanisms provided by Tuxedo (queuing,...). The scalability is realized by exploiting the Tuxedo's server class infrastructure. Figure 8.1. shows the possible architecture of this server. A (Java) Agent or Java code arriving at the agent server (1) is queued as a request for the Agent Execution class. The queued requests are assigned to a process of the Agent Execution class via the standard Tuxedo mechanisms (2). The processes of this server class provide all mechanisms to execute a java agent or a java program: java interpreter, communication mechanisms (3) and access to local services(4). Communication between agents (either on the same location or on other locations) is realized using a dedicated process, the location manager. This process manages all information about the location (available services, agents residing on location, ...) and is responsible for the communication between agents. Agents/Java programs only waiting for an event (communication to another agent, reply from service,....) are moved (5) from their server process to a waiting queue (code, data, state) so that the server process is able to execute the next agent in the request queue. The waiting queue is also managed by the location manager. It has to monitor the events the agents in the queue are waiting for. If an event occurs an agent is waiting for, the agent is placed back in the request queue (6) (with high priority if priorities are available).

The last step will be the implementation of an example application to demonstrate the validity of the concepts and implementations realized in the project (WP 2.4).

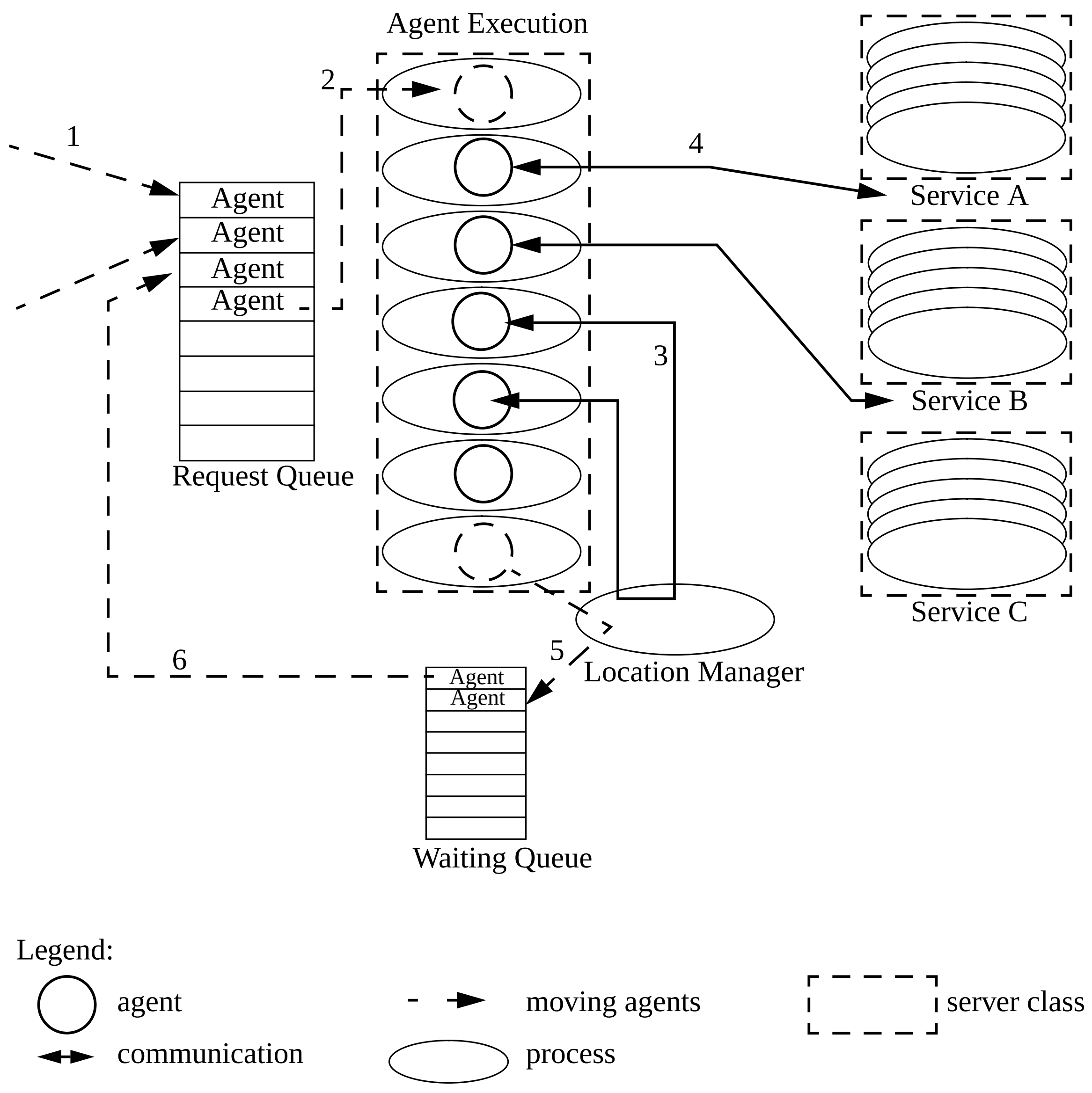


**Figure 8.1.** Agent Server Architecture

# 9 References

**[BA90]** Black, Andrew P.; Artsy, Yeshayahu: Implementing Location Independent Invocation, IEEE Transactions on Parallel and Distributed Systems, Vol 1, No. 1, January 1990

**[Ber97]** Bernstein, P: Principles of Transaction Processing, Morgan KaufmannPubl. Inc., 1997

**[BH+97]** J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel and M. Straßer. Communication Concepts for Mobile Agent Systems. In: Mobile Agents, Proc. 1st Int. Workshop, MA ‘97. Springer, 1997.

**[BN84]** A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls, In ACM Trans Computer Systems, Vol. 2, pp. 39-59. 1984

**[BvR94]** Birman, K.P.; van Renesse, R.: Reliable Distributed Computing with the ISIS Toolkit, IEEE Computer Society Press, 1994

**[Cal97]** Callaghan, Brent: WebNFS - The Filesystem for the Internet, White Paper, Sun Microsystems, 1997
http://www.sun.com/sunsoft/solaris/networking/webnfs/webnfs.html

**[CG89]** Carriero, N.; Gelernter, D.: Linda in Context, CACM 32(4), April 1989

**[Che97]** Cheswick, William R.: Internet Security and Firewalls : Repelling the Wily Hacker, Addison-Wesley, 1997

**[CK97]** T.-H. Chia and S. Kannapan. Strategically Mobile Agents. In: Mobile Agents, Proc. 1st Int. Workshop, MA ‘97. Springer, 1997.

**[CPV97]** A. Carzaniga, G.P. Picco and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. To appear in: Proc. 19th Int. Conf. on Software Engineering, Boston. 1997

**[FMM94]** Finin, T.; McKay, D.; McEntire, R.: KQML as an Agent Communication Language, in: Proc. Third Int. Conf. On Information and Knowledge Management, ACM Press, November 1994

**[GCK96]** Gray, Robert; Cybenko, George; Kotz, David; Rus, Daniela: Agent Tcl. To appear in: Itinerant Agents: Explanations and Examples with CD-ROM, Manning Publishing, 1996.

**[GJS96]** Gosling, James; Joy, Bill; Steele, Guy: The Java Language Specification, Addison-Wesley, 1996

**[GM96]** General Magic, Inc. The Telescript Language Reference., 1996
http://www.genmagic.com-Telescript/TDE/TDEDOCS_HTML/telescript.html

**[GM97]** General Magic, Inc: Odyssey, 1997
http://www.genmagic.com/agents/odyssey.html

**[Gra96]** Gray, Robert: Agent Tcl: A flexible and secure mobile-agent system, in: Proceedings of the Fourth Annual Tcl/Tk

**[HCK95]** C.G. Harrison, D.M. Chess and A. Kershenbaum. Mobile Agents: Are they a good idea? IBM Research Report, 1995

**[Hoh97]** Hohl, Fritz: An Approach to Solve the Problem of Malicious Hosts in Mobile Agent Systems, Bericht Nr. 1997/03, Fakultät Informatik, University of Stuttgart, 1997

**[HS80]** Hammer, Michael; Shipman, David: Reliability Mechanisms for SDD-1: A System for Distributed Databases, in: ACM Transactions on Database Systems 5:4, December 1980

**[IBM95]** IBM Corporation: Open Blueprint Introduction, 1995.
http://www.software.ibm.com./openblue/papers/obintrwb.htm

**[IBM96]** IBM Tokyo Research Labs: Aglets Workbench: Programming Mobile Agents in Java, 1996. http://www.trl.ibm.co.jp/aglets

**[ION96]** IONA Technologies Ltd: OrbixTalk Programming Guide, April 1996

**[JRS95]** Johansen, Dag; van Renesse, Robbert; Schneider, Fred: An Introduction to the TACOMA Distributed System - Version 1.0. Technical Report 95-23, University of Tromso, June 1995

**[KLA90]** Kazar, Leverett, Anderson et. al.: DEcorum File System Architectural Overview, in: Proc. Summer 1990 USENIX Conf., Summer 1990

**[Kla97]** Klar, Peter: Ein verteiltes Serversystem für die Codemigration mobiler Agenten, Diplomarbeit Nr. 1470, Fakultät Informatik, Universität Stuttgart, 1997

**[KMV96]** Konstantas, Dimitri; Morin, Jean-Henri; Vitek, Jan: MEDIA: A Platform for The Commercialization of Electronic Documents, in: Object Applications, ed. Dennis Tsichritzis, University of Geneva, 1996

**[LDD95]** Lingnau, Anselm; Drobnik, Oswald; Doemel, Peter: An HTTP-based Infrastructure for Mobile Agents, Proc. of the 4th International WWW Conference, December 1995.
http://www.w3.org/pub/Conferences/WWW4/Papers/150/

**[LY97]** Lindholm, Tim; Yellin, Frank: The Java Virtual Machine Specification, Addison Wesley, 1997

**[Mae94]** Maes, P.: Agents that Reduce Work and Information Overload, in: CACM 37(7), July 1994

**[Mul90]** Mullender, S.: Distributed Systems, ACM Press, 1990

**[OHE96]** Orfali, R.; Harkey, D; Edwards, J.: The Essential Client/Server Survival Guide, Wiley Computer Publishing, 1996

**[OMG94]** Common Object Services Specification, Volume 1, OMG Document Number 94-1-1, March 1994

**[OMG97]** CORBAservices: Common Object Services Specification, OMG Inc. Publications, 1997
http://www.omg.org/corba/sectrans.htm

**[PD96]** L. Peterson and B. Davie: Computer Networks: A System Approach; Morgan Kaufmann Publishers, Inc.; 1996

**[Pei96]** Peine, H: Ara: Agents for Remote Action. To appear in: Itinerant Agents: Explanations and Examples with CD-ROM, Manning Publishing, 1996.

**[Pei97]** Peine, H: Ara: Agents for Remote Action, in: Cockayne; Zyda: Mobile Agents: Explanations and Examples, Manning Publishing, 1997.

**[RFC1094]** NFS: Network File System Protocol specification., RFC 1094, 1989

**[SBH97]** Straßer, Markus; Baumann, Joachim; Hohl, Fritz: Mole: A Java based mobile agent system, in: Baumann;Tschudin;Vitek (editors): Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems, dpunkt, 1997

**[Sch96]** Schneier, Bruce: Applied Cryptography, John Wiley &Sons, 1996

**[SL95]** Sandholm, T.; Lesser, V.: Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Framework, in: Proceedings of the First International Conference on Multiagent Systems (ICMAS-95), 1995

**[SS97]** Straßer, M.; Schwehm M.: A Performance Model for Mobile Agent Systems. To appear in: Proc. of the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)

**[Sta95]** Stallings, William: Network and Internetwork Security, IEEE Press, 1995

**[Sun96a]** Sun Microsystems: Solaris NEO: Operating Environment Product Overview, March 1996.
http://www.sun.com/solaris/neo/whitepapers/SolarisNEO.front1.html

**[Sun96b]** The JDBC$^{(tm)}$ Database Access API
http://splash.javasoft.com/jdbc

**[Sun96c]** JDBC$^{TM}$ - Connecting Java and Databases
http://java.sun.com/products/jdk/1.1/docs/guide/jdbc/index.html

**[Sun96d]** The Java Transaction Service API
http://splash.javasoft.com/jts/jts.html

**[Tan92]** Tanenbaum, Andrew: Modern Operating Systems, Prentice Hall, 1992

**[Tan95]** Tanenbaum, A.: Distributed Operating Systems, Prentice Hall International, 1995

**[Tan96a]** Introduction to NonStop Transaction Processing, Tandem Manual PartNumber 125335, Tandem Computers Inc., 1996

**[Tan96b]** Safeguard Users Guide, Tandem Manual PartNumber 127299, Tandem Computers Inc., 1996

**[Tsc94]** Tschudin, Christian: An Introduction to the M0 Messenger Language. Technical Report No 86 (Cahier du CUI), University of Geneva, 1994

**[Wa82]** Walter, B.: A Robust and Efficient Protocol for Checking the Availability of Remote Sites, in: Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks, Pacific Grove, February 1982