Universität Stuttgart
Fakultät Informatik

Mole - Concepts of a Mobile Agent System

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

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

Mole - Concepts of a Mobile Agent System

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

Bericht 1997/15
August 1997

Mole - Concepts of a Mobile Agent System

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

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

Abstract

Due to its salient properties, mobile agent technology has received a rapidly growing attention over the last few years. Many developments of mobile agent systems are under way in both academic and industrial environments. In addition, there are already various efforts to standardize mobile agent facilities and architectures.

Mole is the first Mobile Agent System that has been developed in the Java language. The first version has been finished in 1995, and since then Mole has been constantly improved. Mole provides a stable environment for the development and usage of mobile agents in the area of distributed applications.

In this paper we describe implementation techniques for mobility, present communication concepts we implemented in Mole, discuss security concerning Mobile Agent Systems, and present system services provided by Mole.

1 Introduction

Throughout the past years the concept of software agents has received a great deal of attention. Depending on the particular point of view the term ‘agent’ is associated with different properties and functionalities, ranging from adaptive user interfaces, cooperating intelligent processes to mobile objects. Our particular interest lies in the exploration of mobile agents in the Internet and the key benefits provided by the application of this new technology (e.g. in the area of the WWW).

Mobile agents are defined as active objects (or clusters of objects) that have behaviour, state and location. Mobile agents are *autonomous* because once they are invoked they will autonomously decide which locations they will visit and what instructions they will perform. This behaviour is either defined implicitly through the agent code (see e.g. [Gray95]) or alternatively specified by an - at runtime modifiable - itinary (see e.g. [WongEA97]). Mobile agents are mobile since they are able to migrate between locations that basically provide the environment for the agents’ execution and represent an abstraction from the underlying network and operating system.

With the properties printed out above it has been often argued that mobile agents provide certain advantages compared to traditional approaches as the reduction of communication costs, better support of asynchronous interactions, or enhanced flexibility in the process of software distribution. The employment of mobile agents has been particularly promising in application domains like information retrieval in widely distributed heterogeneous open environments (e.g. the WWW), network management, electronic commerce, or mobile computing. The question what real advantages mobile agents offer has been subject of various papers (e.g. [HaChKe95], [BaTsVi96], [RoHoRa97]) and ongoing discussions in mobile agent mailing lists. The results of these investigations and discussions show that we are far from a common understanding concerning the pros and cons of mobile agent technology. But all agree on the following:

To support the paradigm of mobile agents, a system infrastructure is needed, that provides the functionality for the agents to move, to communicate with each other and to interact with the underlying computer system. Furthermore this infrastructure has to guarantee privacy and integrity of agents and underlying system to prevent malicious agents attacking other agents or the computer system. At the same time the agents have to be protected against a malicious system to avoid manipulations of the agents while they visit this system.

In this paper we describe the current state of our agent system infrastructure, the Mobile Agent System Mole V2. Mole builds on Java [Sun97] as the environment for the agent system and as the language for the implementation of the agents.

This paper is organized as follows: after a short introduction into our agent model in section 2 we discuss possible mobility concept for mobile agent in section 3 and present the choice we made for Mole. In section 4 we describe the communication concepts of Mole V2. In section 5 security concerning Mobile Agent Systems is examined. After presenting our notion of agent ids in section 6 we discuss the Mole system services in section 7. In section 8 we give an overview over related work. In section 9 we summarize the paper and present our planned future work.

2 Our Agent Model

In this section we will give only a short overview of our agent model, that has been described in much more detail in [StBaHo96] and [BaumEA97a]. Our model of an agent-based system - as various other models - is mainly based on the concepts of agents and places. An agent system consists of a number of (abstract) places, being the home of various services. Agents are active entities, which may move from place to place to meet other agents and access the places' services. In our model, agents may be multi-threaded entities, whose state and code is transferred to the new place when agent migration takes place. Places provide the environment for safely executing local as well as visiting agents.

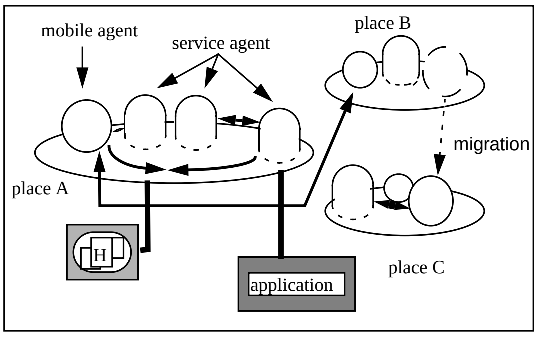


Figure 1: The Agent Model

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

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

3 Mobility Concepts for Mobile Agents

In this section we discuss the different kinds of mobility. We define a taxonomy of the different kinds of mobility, discuss the advantages of the different approaches and present the implementation of mobility in Mole.

3.1 Taxonomy of Mobility

Different degrees of mobility can be distinguished (see figure 2):

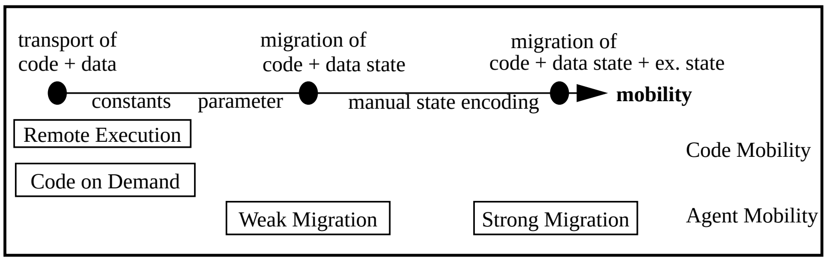


Figure 2: Degree of mobility

In the case of ***Remote Execution***, the agent program is transferred before its activation to some remote node, where it runs until to its termination, i.e., an agent is transferred only once. The information transferred includes the agent code plus a set of parameters (although the transfer of code may not be necessary at runtime, compare e.g. [HoKlBa97] for a discussion of code transport issues). Once activated, an agent itself may use the *Remote Execution* mechanism to start the execution of other agents. A very similar approach - tailored to a client/server-style of interaction - is the *Remote Evaluation* scheme introduced by Stamos [Stamos1986]. With this approach, an operation (e.g. a procedure plus parameters) is transferred to a remote site, where it is performed entirely. After executing the operation, the remote site returns the operation's results back to the issuer of the remote evaluation. The remote evaluation mechanism can be applied recursively, resulting in a tree-structured execution model.

With the above scheme, the destination of the agent program to be transferred is determined by the entity initiating the remote execution. In contrast, with ***Code on Demand***, the destination itself initiates the transfer of the program code. If the *Code on Demand* scheme is used in client/ server settings, programs stored on server machines are downloaded to clients on demand. The currently most popular technologies supporting this type of mobility are ActiveX (see e.g. [AarAar97]) and Java Applets (see e.g. [Sun94]).

Both *Remote Execution* and *Code on Demand* support 'code mobility' rather than 'agent mobility' as both schemes transfer agent programs before their activation. In the following two schemes, agents (i.e., executions of agent programs) may migrate from node to node in a computer network. Obviously, for migrating agents not only code but also the state of the agent has to be transferred to the destination. We will start with *Strong Migration* and then motivate the existence of *Weak Migration*. For our discussion we will assume that an agent's state consists of data state (i.e. the arbitrary content of the global or instance variables) and execution state (i.e. the content of the local variables and parameters and the executing threads).

The highest degree of mobility is *Strong Migration* [GheVig97]. In this scheme, the underlying system captures the entire agent state (consisting of data and execution state) and transfers it together with the code to the next location. Once the agent is received at its new location, its state is automatically restored. From a programmer's perspective, this scheme is very attractive since capturing, transfer and restoration of the complete agent state is done transparently by the underlying system. On the other hand, providing this degree of transparency in heterogeneous environments at least requires a global model of agent state as well as a transfer syntax for this information. Moreover, an agent system must provide functions to externalize and internalize agent state. Only few languages allow to externalize state at such a high level, e.g., Facile [Knabe95] or Tycoon [MaMaSc95]. Since the complete agent state (including data and execution state) can be large - in particular for multi-threaded agents - strong migration can be a very time-consuming and expensive operation.

These difficulties have led to the development of the so-called ***Weak Migration*** scheme, where only **data** state information is transferred. The size of the transferred state information can be limited even more by letting the programmer select the variables making up the agent state. As a consequence, the programmer is responsible for encoding the agent's relevant execution states in program variables. Moreover, the programmer must provide some sort of a *start* method that decides, on the basis of the encoded state information, where to continue execution after migration. This method reduces substantially the amount of state to be transferred. But it changes the semantics of a migration, a fact that every agent programmer has to be aware of.

The following table classifies existing agent systems with regard to the degree of mobility supported.

Table 1: Mobile Agent Systems Classified

Type of Mobility	Systems		
Remote Execution	Java Servlets (push [Sun96])	Remote Evaluation [Stamos86]	Tacoma [JovRSc95]
Code on Demand	ActiveX [AarAar97]	Java Applets [Sun94]	Java Servlets (pull [Sun96])
Weak Migration	Aglets [IBM96]	Mole	Odyssey [GenMag97]
Strong Migration	AgentTcl [Gray95]	Ara [Peine97]	Telescript [GenMag96]

3.2 Advantages of Code and Agent Mobility

In the following, we will examine the advantages resulting from code and agent mobility. During our discussions, for each of the indicated advantages we will point out which degree of mobility is required. We have identified the following prime advantages: Software distribution on-demand, asynchronous operation of tasks, reduction of communication cost, scalability due to dynamic placement of functions.

Software-Distribution on Demand

In existing client-server systems, new code has to be installed manually by users or system operators. The installation is sometimes rather challenging and often requires detailed knowledge about the current state of the used computer system. Finally, software tends to depend on certain versions of other software packages and the installation varies on different machines, operating systems and so on.

With the wide employment of Code on Demand systems (i.e. the success of Java-enabled web browsers) another, easier installation alternative showed up: the *Software-Distribution on Demand* which is able not only to transport code, but also to install packages automatically. For achieving that, code servers offer programs to clients, which include an environment to install these modules. The usage of a platform-independent language like Java allows the system to employ the same installation process on each system and hides differences at the execution of the code. Since software-distribution on Demand is a potential advantage of every mobile code system, mobile agent systems can offer it also, but since the latter uses a less transient model of applications (e.g. the active existence of Java Applets is bound to the existence of the invocation of a browser), code can be installed in a persistent way.

What degree of mobility is needed here? If the client actively calls the code, *Code on Demand* is certainly the matching scheme, but also a kind of *Remote Execution* can be applied if the server e.g. periodically disseminates new versions of a program to registered clients.

Software-distribution on demand only simplifies the management of an existing structure. The following advantages will allow to build up new structures that are better than the old ones in some aspects.

Reduction of Communication Costs

Using agent technology does not reduce communication cost per se. However, in certain situations mobile programs/agents may reduce this cost substantially. Two types of reductions can be distinguished:

- number of (remote) interactions (i.e., between entities residing on different nodes), and
- the amount of data communicated over the network

The first type of reduction can be achieved by bringing two entities that (heavily) interact with each other to the same location. For the second type of reduction consider a client/server relationship, where the client includes a function filtering the data retrieved from the server. The amount of data transferred from the server to the client can be reduced by moving the filter function (or the entire client) to the server. Then, filtering can be done before the data is communicated over the network.

Of course, moving agents is not for free. An overall cost reduction is only obtained if the performance gains exceed the extra overhead for transferring agents. A performance model taking into account an agent’s itinary is described in [StrSch97]. Other models are given in [CaPiVi97] and in [ChiKan97].

Communication cost reductions can already be achieved with the *Remote Execution* scheme. However, agent mobility provides more room for optimizations.

Asynchronous Tasks

Asynchronous communication mechanisms, such as asynchronous message queues (see e.g. [IBM93]), allow asynchronous processing of requests (figure 3). While the individual requests of a task can be processed asynchronously, the client performing this task must be available to receive replies and react on them. Especially in the case of mobile clients or *associated* places this can be problematic. Keeping a mobile client up and connected while task processing is in progress is expensive at least and might even be impossible.

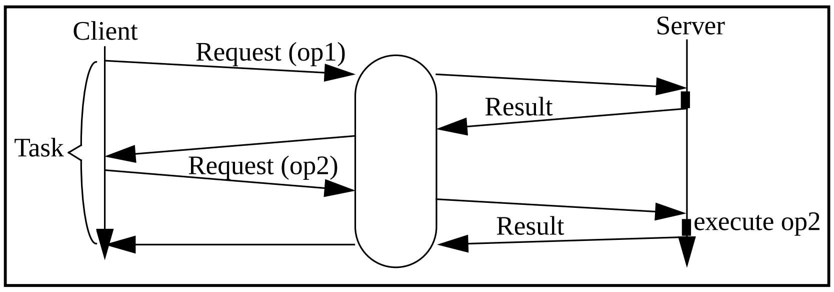


Figure 3: Asynchronous tasks using message queues

With agent technology, the client part of the application can be transferred from the mobile device to stationary servers in the network. From an end user's perspective, not only individual requests but the entire task is moved to the network, where it is performed asynchronously. Clearly, once the task transfer is complete, the mobile device can be disconnected from the network. Later, after days or even weeks, the device can be reconnected to receive the task's results. It is important to notice that the hidden assumption of those scenarios is that the underlying system guarantees 'exactly once' semantics of agents, i.e. when accepting an agent, the network guarantees that the agent is not lost and is performed exactly once, independent of communication and node failures. Unfortunately, none of the current agent systems supports this level of fault-tolerance.

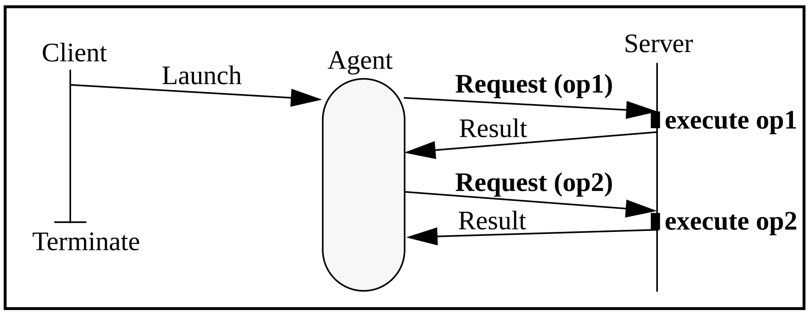


Figure 4: Asynchronous tasks using mobile agents

Now, the question is what degree of mobility is required for those scenarios. *Remote Execution* is sufficient for moving the client program to a *connected* place. This *connected* place (see section 2) is assumed to be an infrastructure component, whose purpose is to host clients downloaded to the network. Downloaded clients run on these places and access remote services to perform their tasks. In general, common servers cannot be used to do this hosting job, e.g., a Lufthansa server is certainly not willing to host an agent booking a flight with a remote BA server. With a *Weak* or *Strong Migration* scheme, hosting agents are not needed anymore. An agent moves from server to server to achieve its task. In our example above, the mobile agent just goes on to the BA server if the Lufthansa server cannot offer the desired flight.

Scalability Due to Dynamic Deployment

Dynamic deployment of agent programs allows for more scalable applications. Assume, for example, a search application that accesses a large number of globally distributed data sources. Assume documents are retrieved from the data sources and selected (or indexed) based on a content-based filtering function. In a pure client/server setting, a client would access the remote data sources, and all retrieved documents would be transferred to the client. The final filtering would be performed at the client site. If accessing the data sources is performed in parallel, the client as well as (parts of) the network may become a bottleneck.

With mobile agents, a hierarchy of filter agents can be set up. Filter agents not only perform content-based filtering but also get rid of redundantly retrieved documents. The structure of the hierarchy and the placement of the individual filter agents mainly depends on the set of data sources accessed. Both placement and structure can change if new data sources are detected while the search operation is in progress. Obviously, this setting is more scalable since filtering is distributed and can be performed close to the data sources. Moreover, redundant information can be detected early and thus does not have to be transported all the way to the client.

What degree of mobility is needed here? *Remote Execution* is certainly sufficient if the placement of (filter) agents is static for a given search operation. If, however, the placement is changed and (filter) agents maintain context information (e.g., in order to detect redundant documents), *Strong* or *Weak Migration* would simplify the implementation.

3.3 Mobility in Mole

As has been discussed above, *Migration*, be it *Weak* or *Strong*, has many advantages. *Remote Execution*, while sufficient for many applications, provides neither the flexibility nor the simplicity in use, that *Weak* and *Strong Migration* supply.

The difference between *Weak* and *Strong Migration* is a change in semantics, but not in expressive power. We have decided not to implement *Strong Migration*, but to choose *Weak Migration* instead. Why? One of the design goals of Mole is the ability to run out of the box on every Java virtual machine (VM). A normal Java VM doesn't support capturing the state of a thread, which would be a prerequisite for capturing the execution state. Thus our decision was to choose the changed semantics and with it the ability to run Mole on unchanged Java interpreters. This includes that, while agents in Mole can be multithreaded, after a migration only one thread is started. If more threads are necessary the agent has to start them explicitly.

```
start()
{
    // here starts the agent thread
    // after migration or at instantiation
    ...
    migrateTo(targetlocation);

    // migration failed if control flow
    // executes the following statements
    ...
}
```

Figure 5: Migration of an agent

Weak Migration in Mole is implemented by using a part of the Remote Method Invocation package RMI, the object serialization provided as part of Java 1.1. After an agent thread calls the migrateTo()-method, all threads belonging to the agent are suspended (not stopped). No new messages and calls (RPC) to the agent are accepted. After all pending messages to the agent have been delivered, the agent is removed from the list of active agents. Now the agent is serialized using the object serialization. The object serialization computes the transitive closure of all objects belonging to the agent (ignoring transient objects and threads), and creates a system-independent representation of the agent. This *serialized* version of the agent is sent to the target location that reinstantiates the agent. If any of the java classes needed are not available locally, the target location requests these classes either from a code server [HoKlBa97], or from the source location. Now the agent is reinstantiated. One new agent thread is started. This thread begins its work at the start()-method. As soon as the thread assumes control of the agent, a success message is sent back to the source location. The source location now terminates all threads pertaining to the agent and removes it from the system.

If at any stage of the migration an error occurs, the migration is stopped and the agent threads at the source location are resumed. The control flow continues after the migrateTo() statement, where error handling can be implemented.

Interestingly, experience showed that the semantics of *Weak Migration* are well understood and easily used even by inexperienced agent programmers. After working with *Weak Migration* for over two years we no longer deem *Strong Migration* necessary, and a large fraction of agent system builders concurs [BaumEA97b].

4 Communication Concepts for Mobile Agents

In this section, we will address the various types of communication suitable for agents and discuss their use in Mole. An in-depth discussion of communication paradigms suitable for agents can be found in [BaumEA97a].

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 (we will discuss in section 6). 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 introduce the concept of a session, which is an extension of Telescript's meeting metaphor [GenMag96]. 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 support both message passing and remote method invocations (with or without session context).

In the general case, a group of agents performing a common task may be arbitrarily structured and highly dynamic. In those environments, we 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 priori which agent or agent subgroup is responsible for generating this event. Therefore, we 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.

4.1 Types of Agent Communication

Considering inter-agent interaction, we distinguish between following types of communication:

- 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.

- 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. Assume e.g. a mobile agent passing a form to another agent and terminating afterwards. 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 [FiMKME94], are based on message passing.
- 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 message sent. 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 themselves 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 true 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 [BirvRe94, KaaTan91]), the concept of tuple spaces [CarGel89], 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. A group model using such a distributed event service for the coordination has been presented in [BauRad97].
- User/Agent Interaction
  Although a very interesting area of research, the interaction between human users and software agents is beyond the scope of this paper. For a discussion of this type of communication the reader is referred to e.g. [Maes94].

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 needed. In our model, message passing and RPC is session-oriented, which means that agents wanting to communicate have to establish a session before they can send and receive data. In the remainder of the section, we discuss the concept of session-oriented communication in the context of Mole and examine event management as a means for anonymous communication.

4.2 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 above and the so-called badges.

Agent_Ids are well-suited for identifying service agents, as long as 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 this system. We will present the directory service Mole provides in section 7.

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 unilaterally terminate the sessions it is involved in at any point in time. Consequently, agents cannot be “trapped” in sessions.
- While an agent is involved in a session, it is not supposed to move to another place. However, if it decides to move anyway, the session is terminated implicitly. The main reason for this property is to simplify the underlying communication mechanism, e.g., to avoid the need for message forwarding.

The question may arise, why sessions are needed at all. There are basically two reasons: Firstly, the concept of a session can be used to synchronize agents that want to ‘meet’ for cooperation. Note that the first property stated above allows agents to ‘meet’ even if they stay on different places. The concept of a session is introduced to allow agents to specify which other agents they are interested to meet at which places. Furthermore, it allows agents to wait until the desired cooperation partner arrives at the place and indicates its willingness to participate.

Secondly, we want to support both “stateless” and “stateful” interactions. In contrast to the first, the latter maintain state information for a sequence of requests. Obviously, if they encapsulate “stateful” servers, service agents have to be “stateful” also. A prerequisite for building “stateful” entities are explicit communication relationships, such as sessions.

Session Establishment

In order to set up sessions two operations are offered, *PassiveSetUp* and *ActiveSetUp*. (see figure 6). The first operation is non-blocking and is used by agents to express that they are willing to participate in a session. In contrast, *ActiveSetUp* is used to issue a synchronous setup request, i.e., the caller is blocked until either the session is successfully established or a timeout occurs.

void **PassiveSetUp**(PeerQualifier , PlaceId)
SessionObject **ActiveSetUp**(PeerQualifier, PlaceId, Timeout)
void cb(SessionObject)
void **SessionObject.Terminate**()

Figure 6: Session Methods

If *ActiveSetUp* succeeds, it returns the reference of the newly created session object to the caller. Input parameter *PlaceId* identifies the place, where the desired session peer is expected, and *PeerQualifier* qualifies the peer at the specified place. A *PeerQualifier* is either an agent_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. In that case a randomly picked agent is chosen. To avoid infinite blocking, the parameter *TimeOut* can be used to specify a timeout interval. The operation blocks until the session is established or a timeout occurs, whatever happens first.

The parameters *PeerQualifier* and *Place_Id* of the operation *PassiveSetUp* are optional. If neither of both parameters is specified, the caller expresses its willingness to establish a session with any agent residing at any place. By specifying *Place_Id* and / or *PeerQualifier* the calling agent may limit the group of potential peers. For example, a group may be limited to all agents wearing the badge “Stuttgart University” and / or that are located at the caller’s place.

As pointed out above, before a session is established both participants must agree explicitly. An agreement for session setup is achieved if both agents issue matching setup requests. Two setup requests, say R_A and R_B of agents A resp. B, match if

- *Place_Id* 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 figure 7). 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

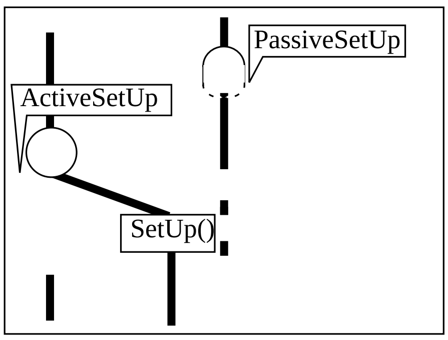


Figure 7: Client / Server Interaction

can be established in parallel, where session establishment is purely client driven.

If both agents issue (matching) *ActiveSetUp* requests this corresponds to a rendezvous. Both requesters are blocked until the session is established or timeout occurs (see figure 8). 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.

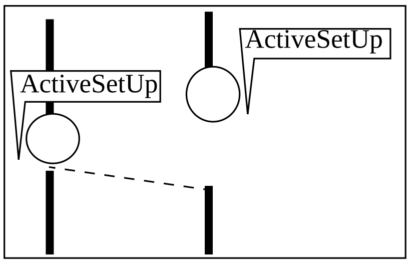


Figure 8: peer to peer interaction

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 9 shows the RMI object enabling access to methods *alpha* and *beta* of object B.

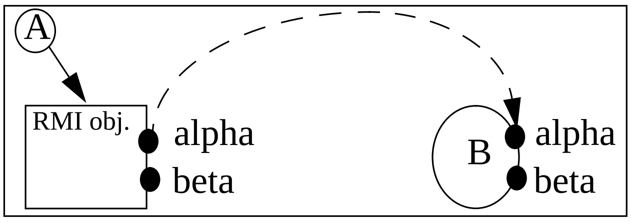


Figure 9: RMI Object

With the ***Messaging object***, messages can be conveyed asynchronously between the participants of a session (see figure 10). 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.

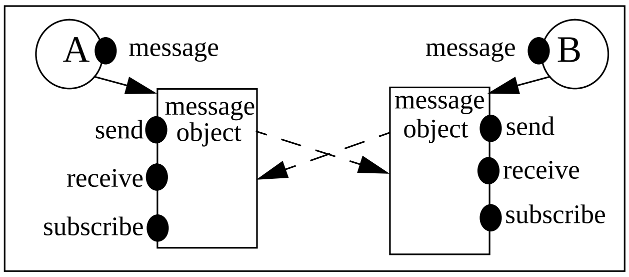


Figure 10: Message Object

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 figure 6), and implicitly when a session participant moves to another place. When a session is terminated, this is indicated by calling the *SessionTerminated* method exported by agents. Moreover, all resources associated with the terminated session are released.

We want to mention, that for easier programming, we still allow the programmer to use “traditional” RMIs or messages without the need of a session overhead, giving them the opportunity to issue single communication acts.

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.

4.3 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 [CarGel89] or [LiDrDo95]. In the remainder of this section 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.

We will now show the suitability of an event service as an infrastructural component at the example of inter-agent synchronization. The general case of using an event service for the inter-agent communication and coordination in agent groups has already been presented in detail in [BauRad97]. First we will define our notion of events, and based on this notion the concept of synchronization objects is presented. We will explore an application scenario, where synchronization is managed through this concept. Finally, a brief overview of the OMG event model is given and the realization of synchronization objects by employing the OMG terminology is described.

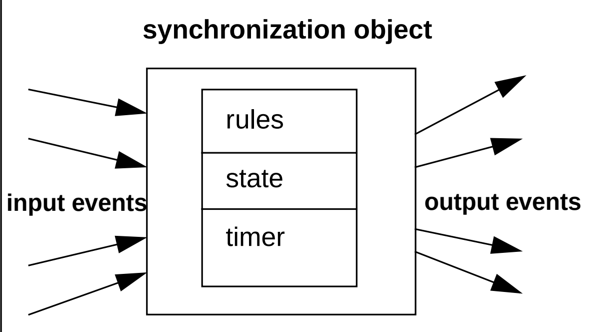
4.3.1 Events

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 actual 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 start-up 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.

4.3.2 Synchronization objects

Synchronization objects (figure 11) are defined as active components responsible for the synchronization of an entire application or 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))
{
      send(event90);
      variable1=false;
}
variable1: boolean = false;
timer1 = after 10 seconds:
{
      send(event51);
}
```

Figure 11: Synchronization 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 in processing 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. Synchronization 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).

4.3.3 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 (figure 12) includes only three event types. The input event *agentsuccess*,

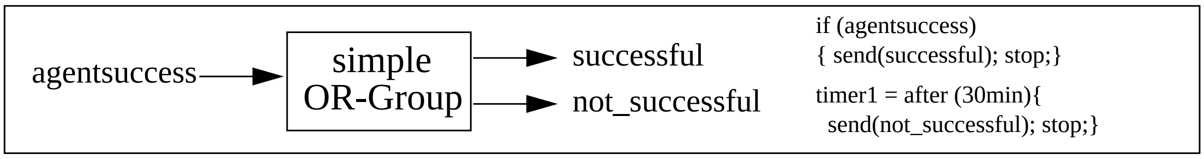


Figure 12: Simple OR Group

signalling the success of an agent, and the output events *successful* and *not_successful*, signalling the group's success. The OR group employs only one rule and one timer. The rule causes the synchronization object to send an event signalling 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.

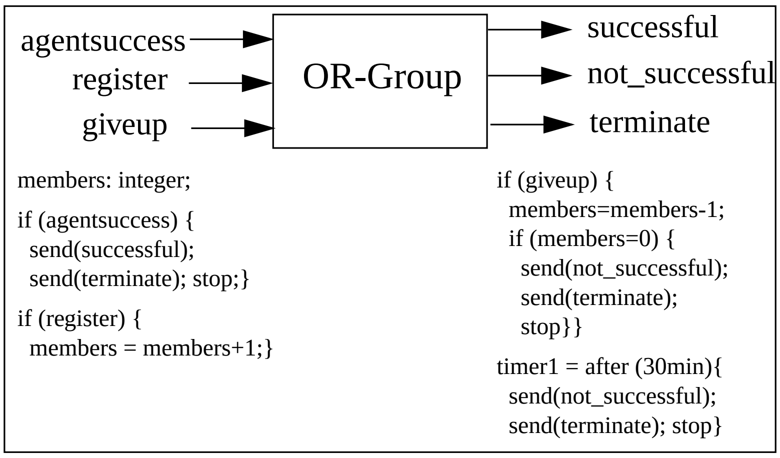


Figure 13: Optimized OR Group

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 figure 13 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.

Alternatively, crash events are used instead of timeout intervals. Crash events are reliable signals that are sent if agents are prevented to terminate their processing successfully (e.g. caused by network partitions, node crashes, or byzantine agent errors). Consequently, if agents can generate success or crash events, no timeout mechanism is needed. However, crash event management is very hard to accomplish. The necessary surveillance protocols are very complex (see [Walter82], [HamShi80]) 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 are well suited 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.

4.3.4 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.

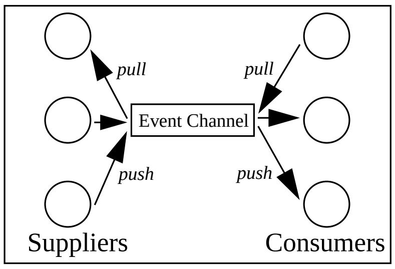


Figure 14: OMG Event Channels

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.

Products following the OMG specification are commercially available (e.g. Iona OrbixTalk[IONA96], or Sunsoft's NEO [Sun96]) or under development (IBM OpenBlueprint [IBM95]).

But none of these products supports mobile participants, a necessary functionality if an event service has to be used with mobile agents. Thus at the University of Stuttgart an event service following the OMG model has been developed that supports mobile participants [Beck97]. This event service builds a hierarchical structure that contains so-called *coordinators* for every participating local network, and in the local network *event demons* for every machine. The *coordinators* communicate via TCP/IP over a minimal spanning tree, that is built dynamically to allow *coordinators* to be added and removed from this communication backbone transparently. The communication with the *event demons* is done via local broadcast, which allows efficient communication. Hand-over of mobile participants is done in a way that guarantees reliable delivery of events.

4.3.5 Synchronization using the OMG model

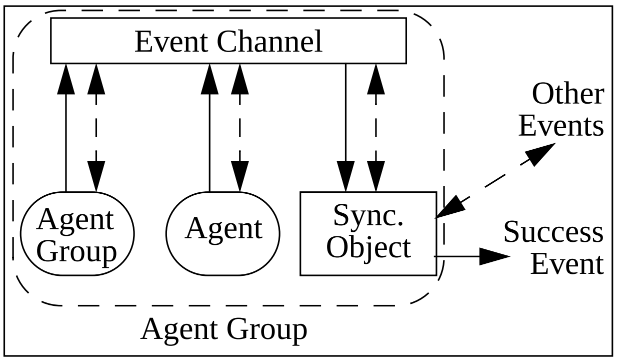


Figure 15: Synchronization using the OMG model

With the employment of an untyped event channel for group communication, OR and AND-groups can be implemented (see figure 15). 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).

5 Security for Mobile Agent Systems

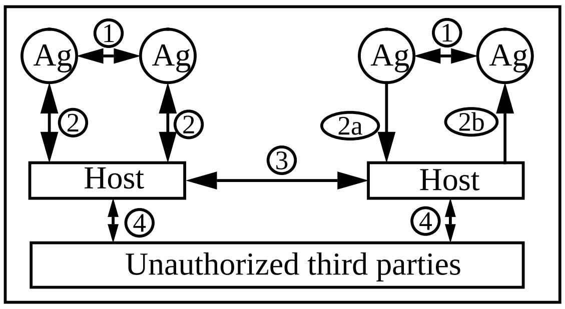


Figure 16: Security Areas of Mobile Agent Systems

The vision of mobile agents as the key technology for future electronic commerce applications can only become reality if all security issues are well understood and the corresponding mechanisms in place. As illustrated by figure 16, four security areas within mobile agent systems can be identified, namely (1) inter-agent security, (2) agent-host security, (3) inter-host security, and (4) security between hosts and unauthorized third parties. Existing cryptographic technology seems to be applicable to areas (1), (3) and (4), but area (2) is specific to mobile code systems (see [Hohl97] for an overview).

The security between host and agent is twofold: on the one side, hosts have to be protected against malicious agents, on the other side, agents have also to be protected against malicious hosts. The first direction, protection against agents, can be solved using existing technology known from Java Applets and SafeTcl programs, since there the same problem exists, the execution of unknown programs. Both systems use an approach, the so-called *Sandbox* security model, where all potential dangerous procedure calls are restricted by special security control components that decide which programs can use these procedures and which not.

The other direction, the protection of agents against malicious hosts, is specific to mobile agents, and ongoing research efforts try to provide approaches in this field. Currently four research directions exist: the organizational approach (as in [GenMag96]) eliminates the problem by allowing only trustworthy institutions to run mobile agent systems (and does, therefore not allow open systems), the trust/reputation approach (see [FaGuSw96] or [RasJan96]) allows agents to migrate only to trusted hosts or such with good reputation (but trust/reputation are problematic terms or they restrict the openness of the system), the manipulation detection approach [Vigna97] offers mechanisms to detect manipulations of agent data or the execution of code (but does not protect against read attacks) and the blackbox protection approach [Hohl97]. This last approach tries to generate a ‘blackbox’ out of agent code by using code obfuscating techniques. Since an attacker needs time to analyse the blackbox code before it can attack the code, the agent is protected for a certain interval. After this ‘expiration interval’, the agent and the data it transports become invalid. All of these approaches are subject of ongoing work, none of them is currently used in real-world application.

6 Agent Ids

In Mole, an agent is seen as a unique entity. This view is supported by using a globally unique name for every agent. This name is immutable, i.e. it does not change when the agent migrates. The uniqueness can only be guaranteed if the system creates the names used. If the system creates the agent ids, then these ids should be devisable without global knowledge. Additionally it is of advantage to be able to derive the site where the agent has been created from the agent id.

Why do we place such constraints on the agent id? First, to be able to identify an agent (this is needed for communication, termination etc.), its name must be unique locally. Second, to be able to do the same after an agent has migrated, the name has to be immutable. From this follows that the agent id has to be globally unique.

This can only be guaranteed if the system itself provides a service to create agent ids conforming to these requirements. If global knowledge is needed to create this agent name, then either a expensive mechanism has to be implemented to obtain the global knowledge, or a single point of failure is introduced if e.g. an id server creating these ids is brought into the system (see e.g the Amoeba sequencer in [Tanenb95]).

The ability to derive the site where the agent has been created is of advantage e.g for finding algorithms utilizing home location registry approach. This approach is used in GSM (see e.g. [MauPau92]), where the id of the user (his telephone number) leads to a designated place (the home location registry) that contains the information how this user can be reached.

The agent id in Mole is created from information that can be obtained locally. Table 2 contains the components of the agent id. The IP v6 address of the underlying system together with the port number of the engine allows to identify the engine on which the agent has been created. The uniqueness of the name is guaranteed by using a combination of a normal counter that is set to 0 at the start of the engine, and a so-called crash counter, that is incremented every time the engine is started. If more than 2^{32} agents are started the crash counter is incremented also. 2 more bytes are reserved for future use, giving a total of 24 bytes.

Table 2: The components of the Agent Id

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

7 Mole System Services

In this section we describe the Mole system services that provide functionality needed on different levels, from the communication and migration level, to the level of system and user agents. First we discuss the naming of agents used in Mole, and present the directory services. Then security mechanisms for Mole are discussed, a service to find agents and resource management for Mole is presented.

7.1 Security

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

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

7.2 Directory Service

A directory service is an electronic database that contains information on entities. An example for a directory service is X.500 (see e.g. [Chadwi94]). In our Mole system we provide a simple local directory service, that provides information on agents providing a service denoted by a string. This local directory service exists on every place.

An agent can register itself locally if it provides a service by submitting a string identifying the service to the directory service.

Another agent wanting to use this service first asks the directory service. The directory service returns a list containing all agents providing the service. This list is either empty, or contains one or more agent ids. The agent now chooses one of the agent ids and contacts the agent.

7.3 Resource Management

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

- CPU time
- local network communication

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

The CPU time used is calculated by counting the time slices given to threads of an agent. Mole has a central object, the MCP (Master Control Process), that schedules all threads in the Mole system. We decided to implement our own scheduler, when problems with Java 1.02 lead to the conclusion that the java scheduler of the solaris implementation had problems with more than 3 threads of the same priority.

The network communication is an important cost factor. Thus it is important for both accounting and resource control. Because all agent communication has to use the mechanisms provided by the agent system, control is done here.

When an agent arrives at one place the arrival time is noted down. This way the total time at the local place can be computed without problems.

8 Related Work

In table 3 an overview over available mobile agent systems is given. Many of these agent systems are research prototypes, and only a few of these have users outside their own university or research institute. We have already classified most of these systems regarding mobility support (see table 1), now we will examine the systems on the subject of communication support.

Table 3: Mobile Agent Systems and their Availability

Name of the System	Supported Languages	Company	Availability
ARA	Tcl, C, Java	University of Kaiserslautern, Germany	free
ffMAIN	Tcl, Perl, Java	University of Frankfurt, Germany	no
Tacoma	Tcl, C, Python, Scheme, Perl	Cormell (USA), Tromso, Norway	free
AgentTcl	Tcl	Dartmouth College, USA	free
Aglets	Java	IBM, Japan	binary only
Concordia	Java	Mitsubishi, USA	binary only
CyberAgents	Java	FTP Software, Inc., USA	no longer
Java-2-go	Java	University of California at Berkeley, USA	free
Kafka	Java	Fujitsu, Japan	binary only
Messengers	M0	University of Zurich, Switzerland	free
MOA	Java	The Open Group, USA	no
Mole	Java	University of Stuttgart, Germany	free
MonJa	Java	Mitsubishi, Japan	binary only
Odyssey	Java	General Magic, USA	binary only
Telescript	Telescript	General Magic, USA	binary only
Voyager	Java	ObjectSpace, Inc., USA	binary only

All of these systems for mobile agents employ many communication mechanisms such as messages, local and remote procedure calls or sockets, but, to our knowledge, no system uses a global event management for communication and synchronization. There are “events” in AgentTcl [GrayEA96], 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 [GenMag96], 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” exception. To summarize the Telescript meeting, we can say, that it is not a session according to our definition.

There are also “meetings” in ARA[Peine96] 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.

9 Conclusion and Future Work

In this paper we presented the Mobile Agent System Mole and the design decisions that led to the existing implementation of Mole V2. Different kinds of mobility have been discussed, *Remote Execution*, *Code on Demand*, *Weak Migration* and *Strong Migration*, and their advantages and disadvantages have been discussed. We gave the reasons for choosing *Weak Migration* in Mole. The communication concepts implemented in Mole have been presented, namely sessions, badges, and event services for mobile participants. Security problems in Mobile Agent Systems have been reviewed, the structure of agent ids as used in Mole have been presented, and the system services of Mole have been discussed.

Since its beginning the system had external active users, that give us feedback (i.e. bug reports, new features etc.) and thus help to improve the system. These are Siemens GmbH, Tandem Computers, University of Freiburg, University of Zurich, and University of Geneva.

Mole is used, among other things, as the infrastructure for an electronic documents system [KoMoVi96], as a simulation environment for distributed network management, as an environment for an enhanced WWW server, as an execution environment for server classes [StraEA97], and in a distributed variant of a Multi-User Dungeon (MUD), in which players can use mobile agents as artificial team-mates.

Apart from these active users over 400 different persons downloaded the version 1.0 of Mole, and in the 2 months since the new version of Mole has been released, already nearly 200 downloads have been counted.

Mole is available as source code. Further informations about the Mole project can be found at http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html.

The next problems we will investigate are group models, a performance model and the area of commercial applications.

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 [GenMag96, HaChKe95]. 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 performance model could provide help to identify situations in which agent migration is advantageous compared to remote procedure calls.

A considerable part of today’s commercial applications require a high degree of robustness. An *exactly-once* semantics for task processing will be a prerequisite for a majority of future net-based applications. Those semantics require a tight integration of agent technology and transaction management. There are two challenges to achieve this integration. Firstly, due to their asynchronous nature, agents are best suited to be engaged in long-lived activities. It must be investigated, which transaction models meet these requirements, where Contracts [WaeReu92] and Sagas [GarciaEA91] seem to be a good starting point. Secondly, at least parts of the agent state must be made recoverable. Current approaches to integrate mobile entities and transactions, such as Java Database Connection (JDBC, see [HaCaFi97]) and Java Transaction Service (JTS, see [JavaSoft97]), only consider server state to be recoverable. If operations performed on mobile state are part of transactions existing protocols for transaction management, such as commit protocols, must be modified.

Furthermore we will continue to investigate agent group models, agent security issues and advanced communication concepts.

Acknowledgements: A system as large as this cannot be implemented with only a few researchers. We wish to thank the many students that have implemented parts of the system as their student or diploma thesis.

A References

[AarAar97] B. Aaron, A. Aaron. “ActiveX Technical Reference”, Prima Pub, 1997.

[AgSoc97] The Agent Society. “The Agent Society Web Page”, 1997, URL: http://www.agent.org/

[BaumEA97a] J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, M. Straßer. “Communication Concepts for Mobile Agent Systems”, in Proc. Mobile Agents ‘97, Springer Verlag, 1997.

[BaumEA97b] J. Baumann, M. Shapiro, C. Tschudin, J. Vitek. “Mobile Object Systems: Workshop Summary”, Workshop Proceedings for the third ECOOP Workshop on Mobile Object Systems, 1997, to appear.

[Bauman97] J. Baumann. “A Protocol for Orphan Detection and Termination in Mobile Agent Systems“, Technical Report Nr. 1997/09, Faculty of Computer Science, University of Stuttgart, 1997.

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

[BaTsVi96] J. Baumann, C. Tschudin, J. Vitek. “Mobile Object Systems: Workshop Summary”, Workshop Proceedings for the 2nd Workshop on Mobile Object Systems, in Workshop Reader ECOOP ’96, dpunkt, 1997.

[Beck97] B. Beck. “Terminierung und Waisenerkennung in einem System mobiler Software-Agenten” (german), Diploma Thesis, Faculty of Computer Science, University of Stuttgart, 1997.

[BirvRen94] K. P. Birman, R. van Renesse. “Reliable Distributed Computing with the ISIS Toolkit”, IEEE Computer Society Press, 1994.

[CaPiVi97] A. Carzaniga, G. Picco, G. Vigna. “Designing Distributed Applications with Mobile Code Paradigms”, To appear in Proc. 19th Int. Conf. on Software Engineering, Boston, 1997.

[Chadwi94] D. Chadwick. “Understanding the X.500 Directory”, Chapman & Hall, 1994.

[ChiKan97] T.-H. Chia, S. Kannapan. “Strategically Mobile Agents”, in Proceedings of the First International Workshop on Mobile Agents, MA’97, Springer Verlag, 1997.

[CarGel89] N. Carriero, D. Gelernter. “Linda in Context”, CACM 32(4), April 1989

[FaGuSw96] W. Farmer, J. Guttmann, V. Swarup. “Security for Mobile Agents: Authentication and State Appraisal”, in: Proceedings of the European Symposium on Research in Computer Security (ESORICS), 1996.

[FiMKME94] T. Finin, D. McKay, R. McEntire. “KQML as an Agent Communication Language”, in Proc. Third Int. Conf. On Information and Knowledge Management, ACM Press, November 1994.

[FIPA97] FIPA. “Foundation for Intelligent Physical Agents”, 1997.
URL: http://drogo.cselt.it/fipa/

[GarciaEA91] H. Garcia-Molina, D. Gawlick, J. Klein, et al. “Modeling Long-Running Activities as Nested Sagas”, Data Engineering Bulletin 14(1): 14-18 (1991) .

[GenMag96] General Magic, Inc. “The Telescript Language Reference”, 1996. URL: http://www.genmagic.com/Telescript/TDE/TDEDOCS_HTML/telescript.html

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

[GheVig97] C. Ghezzi, G. Vigna. “Mobile Code Paradigms and Technologies: A Case Study”, in Proc. Mobile Agents ‘97, Springer Verlag, 1997.

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

[Gray95] R. S. Gray. “AgentTcl: A Transportable Agent System”, Proc. CIKM’95 Workshop on Intelligent Information Agents, 1995.

[GrayEA96] R. Gray, G. Cybenko, D. Kotz, D. Rus. “Agent Tcl.”, Itinerant Agents: Explanations and Examples with CD-ROM, Manning Publishing, 1996.

[HaCaFi97] Hamilton, Cattell, Fisher. “JDBC Database Access with Java”, JavaSoft Press, Addison-Wesley, to appear.

[HamShi80] M. Hammer, D. Shipman. “Reliability Mechanisms for SDD-1: A System for Distributed Databases”, in: ACM Transactions on Database Systems 5:4, December 1980.

[HaChKe95] C. Harrison, D. Chess, A. Kershenbaum. “Mobile Agents: Are they a good idea?”, IBM Research Report, IBM T.J. Watson Research Center, 1995.

[Hohl97] F. Hohl. “An approach to solve the problem of malicious hosts”, Technical Report Nr. 1997/03, Faculty of Computer Science, University of Stuttgart, 1997.

[HoKlBa97] F. Hohl, P. Klar, J. Baumann.”Efficient Code Migration for Modular Mobile Agents”, accepted Submission for the Third ECOOP Workshop on Mobile Object Systems: Operating System support for Mobile Object Systems, 1997.

[IBM93] IBM Corp. “Messaging and Queuing Technical Reference”, SC33-0850, 1993.

[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

[IONA96] IONA Technologies Ltd. “OrbixTalk Programming Guide”, April 1996.

[JavaSoft97] JavaSoft, Inc. “The Java Transaction Service API. Web Page.” URL: http://splash.javasoft.com/jts/jts.html

[JoVRSc95] D. Johansen, R. van Renesse, F. Schneider. “An Introduction to the TACOMA Distributed System - Version 1.0”, Technical Report 95-23, University of Tromso, 1995.

[KaaTan91] M. F. Kaashoek, A. S. Tanenbaum. “Group Communication in the Amoeba Distributed Operating System.”, In Proceedings of the 11th Conference on Distributed Computing Systems, 1991.

[Knabe95] F. Knabe. “Language Support for Mobile Agents”, PhD dissertation, School of Computer Science, Carnegie Mellon University, 1995.

[KoMoVi96] D. Konstantas, J.H. Morin, J. Vitek. “MEDIA: A Platform for The Commercialization of Electronic Documents”, in: Object Applications, ed. Dennis Tsichritzis, University of Geneva, 1996.

[LiDrDo95] A. Lingnau, O. Drobnik, P. Doemel. “An HTTP-based Infrastructure for Mobile Agents”, Proc. of the 4th International WWW Conference, December 1995. URL: http://www.w3.org/pub/Conferences/WWW4/Papers/150/

[Maes94] P. Maes. “Agents that Reduce Work and Information Overload”, in CACM 37(7), July 1994.

[MaMaSc95] B. Matthiske, F. Matthes, J. Schmidt. “On migrating threads”, in Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, 1995.

[MauPau92] M. Mauly, M. Paulet. “The GSM System for mobile Communication.”, Europe Media Publications S. A., 1992.

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

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

[OMG97] Object Management Group. “Mobile Agent Facility (MAF) specification”, http://www.omg.org/library/schedule/CF_RFP3.htm

[Peine96] H. Peine. “Ara: Agents for Remote Action.” In *Itinerant Agents: Explanations and Examples with CD-ROM*, Manning Publishing, 1996.

[RasJan96] L. Rasmusson, S. Jansson. “Simulated Social Control for Secure Internet Commerce”, Accepted Position Paper to the New Security Paradigms ’96 Workshop. URL: http://www.sics.se/~lra/nsp96/nsp96.html

[RoHoRa97] K. Rothermel, F. Hohl, N. Radouniklis. “Mobile Agent Systems: What is Missing?“, in Proc. DAIS ’97, to appear.

[Stamos86] J. W. Stamos. “Remote Evaluation”, TR-354, MIT, 1986.

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

[StraEA97] M. Straßer, J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, M. Schwehm, “ATOMAS: A Transaction-oriented Open Multi Agent-System. Annual Report”, Technical Report Nr. 1997/14, Faculty of Computer Science, University of Stuttgart, 1997.

[StrSch97] M. Straßer, M. Schwehm. “A Performance Model for Mobile Agent Systems”, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA’97, Las Vegas 1997.

[Sun94] Sun Microsystems. “The Java Language: A White Paper”, Technical Report, Sun Microsystems, 1994.

[Sun96] Sun Microsystems. “Solaris NEO: Operating Environment Product Overview”, March 1996.
http://www.sun.com/solaris/neo/whitepapers/SolarisNEO.front1.html

[Sun97] The Java Web Pages. URL: http://www.javasoft.com

[Tacoma97] Tacoma Project Pages. http://www.cs.uit.no/DOS/Tacoma/index.html

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

[Vigna97] G. Vigna. “Protecting Mobile Agents through Tracing”. Accepted Submission for the Third ECOOP Workshop on Mobile Object Systems: Operating System support for Mobile Object Systems, 1997.

[WaeReu92] H. Wächter, A. Reuter. “The ConTract Model”, in Transaction Models (ed. A. Elmagarmid), Morgan Kaufmann, 1992.

[Walter82] B. Walter. “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.

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

[White94b] J. E. White. “Telescript Technology: Scenes from the Electronic Marketplace”, General Magic, 1994.

[WongEA97] D. Wong, N. Paciorek, T. Walsh. “Concordia: An Infrastructure for Collaborating Mobile Agents”, in Proceedings of the First International Workshop on Mobile Agents, MA’97, Springer Verlag, 1997.