Communication Concepts for Mobile Agent Systems[1]

Joachim Baumann, Fritz Hohl, Nikolaos Radouniklis,
Kurt Rothermel, Markus Straßer

Institute of Parallel and Distributed High-Performance Systems (IPVR),
University of Stuttgart, Germany

{baumann,hohlfz,strasser,nsradoun,rothermel}@informatik.uni-stuttgart.de

Abstract. Driven by the question how to identify potential communication partners and the need for well-suited communication schemes in agent-based systems, we discuss two communication concepts: sessions and global event management.

Sessions establish either actively or passively a context for inter-agent interactions. Communication partners are addressed by globally unique agent identifiers or via badges. Communication in sessions is based on RPC or message mechanisms.

Global event management addresses the need for anonymous communication. Event managers are employed as a synchronization means within agent groups. Based on this approach, we introduce synchronization objects, - active components that offer various synchronization services. The presented model is finally mapped onto OMG event services.

1 Introduction

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.

Though first prototype systems (e.g, see [IBM96]) and even products (e.g., see [GM96]) exist, the architecture of mobile agent systems is not well understood today and hence needs more investigation. In our paper, we will address two issues, communication and synchronization in agent-based systems.

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,

1. This work was funded by Tandem Cupertino and the German Research Community (DFG)

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. Numerous existing agent systems are purely based on an RPC-style communication. While this type of communication is mainly 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.

The remainder of the paper is structured as follows. In Section 2, we present an overview of the employed agent model. Various agent communication types and their need for well-suited communication schemes are then discussed in Section 3. Section 4 examines one of these schemes, the session-oriented communication and its benefit for mobile agent systems. Section 5 focuses on global event management as a vital infrastructural component. A brief overview of Mole, our current agent system is presented then. The paper concludes with a list of related work and a summary of the article's key issues and future work.

2 An Agent System: A Collection of Agents and Places

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.

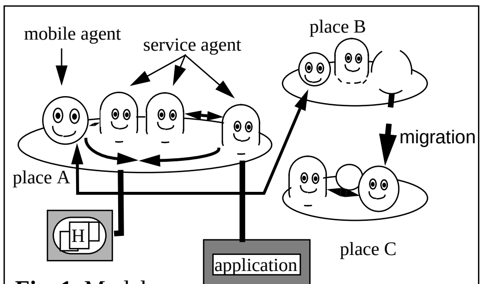


Fig. 1. Model

Our model distinguishes between mobile agents and so-called service agents. Service agents are stationary and interface the services available at places. Those services may include system services, such as a file or directory access, as well as application-level services, such as a hotel reservation or flower delivery service. Service agents encapsulate arbitrary services and represent them in the agent world. From a technical point of view, service agents map the service request expressed in the „agent language“ to the individual service interface. This mechanism allows legacy systems to be incorporated. Moreover, service agents will be the place, where access control mechanisms are located. In contrast to service agents, mobile agents may migrate from node to node.

Each agent, whether mobile or stationary, is identified by a globally unique agent identifier. An agent’s identifier is generated by the system at agent creation time. It is independent of the agent’s (current) location, i.e. it does not change when the agent moves to a new location. In other words, the applied identifier scheme provides location transparency.

A place is entirely located at a single node of the underlying network, i.e. all service agents associated with a place reside on the same node. Conversely, 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, implementing a certain prizing policy, and so on.

The requirement of having places being realized on a single node does not mean that all service implementations have to be located at the place’s node also. It is well conceivable that the place’s service agents provide access not only to local but also to remote services, typically accessible via a LAN. Single node places lead to well defined properties in terms of communication: Intra-place communication between an agent and a service agent is always local an hence is (in general) cheaper than inter-place communication. Moreover, single node places are much simpler to implement than distributed places.

3 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. The communication patterns that may occur in this type of interaction might not be limited to request/response only. The required degree of flexibility 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 paper. 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 the paper, we discuss the concept of session-oriented communication in the context of agent-based systems and investigate event managers for anonymous communication.

4 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 Section 1 and so-called badges.

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: First, 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 at different places. The concept of a session is introduced to allow agents to specify designated agents they are interested to meet at designated 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). 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)
```

Fig. 3. 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 specifying 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 explicitly. An agreement for session setup is achieved if both agents issue matching setup requests. Two setup requests, say R_A and R_B of agents A respectively B, match if

- *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. 4). The agent playing the server role once issues PassiveSetUp when it is ready to receive requests. When an agent playing the client role invokes ActiveSetup, this causes the SetUp method of the server side to be invoked implicitly. SetUp implicitly establishes a session with the caller and assigns a thread for handling this session. Therefore, once the server agent has called PassiveSetup, any number of sessions can be established in parallel, where session establishment is purely client driven.

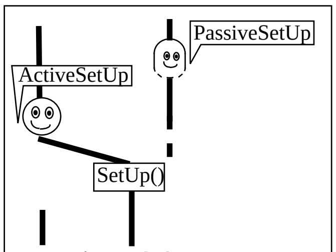


Fig. 4. C-S interaction

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. 5). 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.

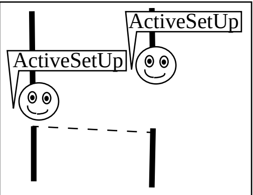


Fig. 5. P-P interaction

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. With the *Messaging* ***object***, messages can be conveyed asynchronously between the participants of a session. Messages are sent by calling the send method. For receiving messages the receive and subscribe methods are provided. The receive method blocks until a message is received or timeout occurs, whatever happens first. If the *subscribe* method is invoked instead, the incoming messages are handed over by calling the *message* method of the recipient and passing the message as method parameter.

The advantage of having the concept of com objects is twofold. First, only those communication mechanisms have to be initiated that are actually needed during a session, and secondly, additional 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), 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.

5 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. In the remainder of this section we will concentrate on event mechanisms as a well-suited concept for inter-agent synchronization.

Applications can be modeled 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 modeling 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.

5.1 Events

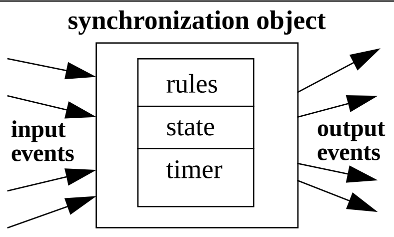
In our notion, events are objects of a specific type, containing some information. Events are generated by so-called producers and are transferred to the consumer by the event service. Consumers (and, depending on the concrete implementation of the event service, also producers) have to register at the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

5.2 Synchronization objects

Synchronization objects 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.

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.



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

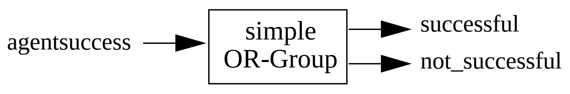
Fig. 6. sync. object

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).

Example: OR and AND groups

Two agent group types of particular interest are the OR-group and the AND-group. AND-groups succeed only if all agents have accomplished their task. 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. 7) includes only three event types. The input event *agentsuc-*



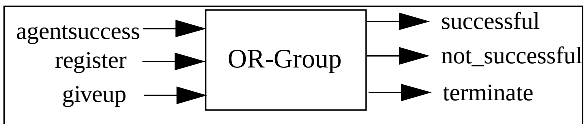
```
if (agentsuccess) then{
  send(successful); stop;}
timer1 = after (30min){
  send(not_successful); stop;}
```

Fig. 7. Simple OR group

cess, signaling 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. 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 an 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}
```

Fig. 8. OR group

5.3 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.

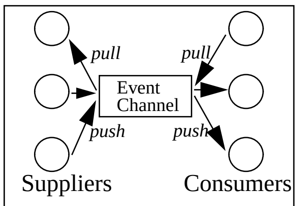


Fig. 9. Event Channels

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[ION96], or Sunsofts NEO [Sun96]).

5.4 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.

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).

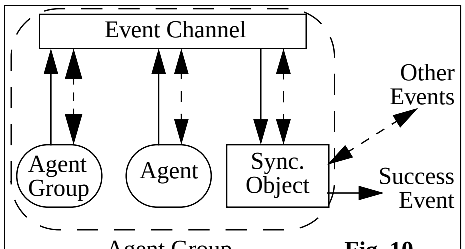


Fig. 10.

6 Mole

In order to allow research in the field of mobile agent systems, Mole [SBH96] was developed at the University of Stuttgart. Mole is a platform for mobile agents which uses Java as the agent programming and as the implementation language. Therefore, Mole is a pure Java application and can be started at every computer platform for which a Java Development Kit is available. Agents in Mole may use multiple threads, have globally unique names and can provide or request services, which can be looked up locally. A service is currently the implementation of one or more methods with specified names and parameters (which are called interfaces in Java), services are requested by calling those methods by using a kind of remote method invocation (a Java object RPC). When it comes to communication, Mole currently supports the (global) exchange of messages and the mentioned RMI. Both are addressed by using a direct addressing scheme. The only way to obtain the needed referencing addresses is the use of the local service look-up mechanism which associates a list of agent names to service names. A mechanism which is able to associate current location to agent names is in the implementation stage. Mole will be used, among other things, as the infrastructure for an electronic documents system [KMV96] and in a distributed variant of a Multi-User Dungeon (MUD), in which players can use mobile agents as artificial team-mates.

Mole is available as source code; the first public version was released in June 1996. Further informations about the Mole project can be found at *http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html*.

7 Related Work

Current mobile agent systems 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. Thus, according to our definition the Telescript meeting is not a session.

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.

8 Summary and 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.

After presenting a brief description of our agent model, we identified different types of communication schemes that please the requirements of agent based systems. 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 synchronization 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. The proposed mechanisms are not implemented yet in our Mole system (see Section 6). Future work will encompass the integration of the session concept and a distributed event service into Mole.

9 Literature

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

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

[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.

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

[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

[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

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

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

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

[SBH96] 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, 1996

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