Java-based Mobile Agents — How to Migrate, Persist, and Interact on Electronic Service Markets[1]

B. Liberman, F. Griffel, M. Merz, W. Lamersdorf
Hamburg University - Department of Computer Science- Distributed Systems Group
[1liberma | griffel | merz | lamersd] @ informatik.uni-hamburg.de

Abstract

This paper presents a mobile agent approach that aims at satisfying the following requirements of open Internet-based electronic service markets: the mobile agent system should be usable by any Internet user without a need for specifically configurated non-standard software tools. It should reduce costs in mobile computing environments and therefore enhance overall efficiency. It should suit well to an electronic service market where local services are commercially offered and business transactions predominate the interaction between customers and suppliers.

As a part of the project OSM (Open Service Model), mobile agents are built on top of two well-established technologies: CORBA and Java. The first is used as a conceptual framework for interoperability, the latter as the programming environment. Since Java does not provide the necessary persistency of execution state, the concept of *OSM service profiles* is used to embed Java classes and to transfer a coarse-grained execution context in a secure and efficient manner.

1 Introduction

Electronic service markets allow customers and suppliers to exchange services against payment through business transactions [Mer96, Schm93]. It is assumed that both individual and standardized services are offered in an electronic service market. However, to satisfy evolution as one of the most fundamental requirement for the market mechanism, the ability to gain ad-hoc visibility and availability — despite of a possibly complex service offer — is an important precondition for service suppliers.

The OSM (Open Service Model) architecture aims at integrating the following mechanisms that allow to perform business transactions in a flexible way:

- Generic user access to remote services is provided by a browsing tool, called the *generic client* [MML94]. It supports customers to establish sessions with suppliers, to integrate them visually at the desktop-level, and to store, transfer, and resume sessions on different network sites.
- The ad-hoc *configuration of supporting services* such as payment, authentication, or notary services. Each time a business transaction is to be established this con-

---

[1] Published in Rothermel/Popescu-Zeletin: „Mobile Agents“, Proceedings MA97 Workshop, Berlin, Springer LNCS #1219 1997

figuration takes place using a unified description technique and a matching mechanism to specify the transaction partners' needs.

- *Value chains* emerge in the following two ways: first, through the establishment of mediators, i.e. services that provide service references to their clients. Mediators may either supply a query interface (such as in the case of the trading service [MML94]) or a browsing interface (on-line catalogues or directories). Secondly, value chains may emerge by enriching, combining, or coordinating existing services.
- The *service profile* is established as a common vehicle for service offer description and as a persistent data store. This allows all involved OSM components to dynamically provide or obtain specific information in a well-structured way: traders may process service type definitions, catalogues may extract icons and description texts, or the generic client may obtain information on the support service requirements of the transaction partner [MTL96].
- Finally, the OSM architecture aims at supporting *service negotiation* between client and server by structuring conversations into a limited set of speech-acts [CFF+92]. This helps to reduce the complexity that is principally given when two parties agree step-by-step on a set of service attributes and allows for a higher level of automation.

It has been discussed in [MML96] that *mobile agents* fit to this architecture if they are considered as value-added service providers: they are developed either by a service supplier or a third party in order to utilize the call interface of a single server or a set of servers that is necessary for a distinct task to be accomplished. To give an example, information retrieval at remote database servers may be performed by a single migrating agent visiting all appropriate database sites. Then, from a conceptual point of view, the mobile agent provides to the customer the independence from server-specific interfaces and semantics. It acts as a value-added service since the possibly complex data query interfaces may be reduced to a simple user entry form that allows only to enter some keywords. Also the possible heterogeneity of different database servers may be hidden by the agent in this way. However, in contrast to an immobile value-added server, the agent is able to actively interact with other agents to solve the given problem cooperatively.

Another important domain which promotes appliance of mobile software appears in connection with mobile computing. The nature of such an environment entails some questions mobile agents are tailored to cope with. Concerning communication costs particularly, the advantage of mobile agents lies in the minimizing of the on-line time and therefore cost reduction.

The rest of this paper is organized as follows: section 2 gives an overview of the design space for mobile agent systems and motivates the selection of Java as the implementation language. Possible problems of this decision are discussed and the service profile concept is introduced as a possible approach for their circumvention. Section 3 focuses on the chosen mobile agent architecture and some of its technical details. A concise example in Section 4 illustrates the integration of the agent system within the

overall OSM architecture's mechanisms in a typical scenario. The outlook summarizes the resulting system and gives a perspective for future research directions.

2 Design choices for agent systems

Mobile agents are alleged to provide suitable techniques for the implementation of electronic market systems that allow both demanders and suppliers of services and goods to exchange them freely based on electronic contract settlement and execution [CGH+95, MRK96]. This application domain determines the requirements which are imposed on agents.

Definition:

In the following, we define a *mobile agent* as an encapsulation of *code*, *data*, and *execution context* that is able to *migrate autonomously* and *purposefully* within computer networks *during execution.*

An agent is able to react sophisticatedly on external events. It may be persistent in the sense that it can suspend execution and keep local data in stable storage. After resuming activity, an agent's execution is continued — but not necessarily at the same location.

Designing agents

In the following, the term *persistent execution state* — as a postulated basis for migration technology — means that the agent should comprise a control flow definition at a coarse level of granularity. This allows not only persistent programming language systems to be capable of representing mobile agents but also conventional ones, which may only be able to provide object-level persistency. These objects may control execution after a migrated agent has been revived at the target host. It must be noted that this relaxation does not really impose any constraints on the agent functionality since a reasonable activity usually comprises at least one method invoking — a high level operation.

The possible spectrum for mobile agent implementations therefore spans:

1. Persistent programming languages such as *Smalltalk* (with a persistent image), *pJava* [Jor96], or *Telescript* [Whi94]. Here, an abstract machine is used to execute portable bytecode that may be transferred forth and back within the network. A fine-grained migration capability is given since the agent may initiate migration at any point of execution (at the *micro-level*). However, persistent programming languages usually require support by databases or object stores, which, in turn, represent a language-specific environment that is not ubiquitously given on each computer in the network.
2. To factor out control flow definitions, workflow-management systems use petri nets or finite automatons, etc. (at the *macro level*). No local processing is done by the agent, which is, in fact, a data structure that is interpreted by each (local) engine in order to invoke a coarse-grained function at the respective network site. This approach does not require any sophisticated language support, however, the

main disadvantage lies in the necessity to factor-out any program logic to the stationary server — even simple arithmetic operations.

3. A well-balanced compromise of the previous two extremes seems to blend the advantages of mobile code with those of persistent data structures that determine control flow. In this case, local operations can still be performed efficiently by the agent code, but migrations will only be possible from distinct migration hooks that define the next entry point and indicate to the executing engine that a transfer of the agent is requested.

In contrast to *language-level persistence* (as given, e.g., in the case of pJava), persistence in Java is currently restricted to selected data objects that are manipulated through dedicated library functions. For this reason, this approach is called *library-level persistence*.

In the following, the third approach will be supported on the basis of Java. As a portable interpreted language, Java allows dynamical loading and binding of classes. The importance of this feature in the agent system environment is derived from the requirement of ad-hoc providing of individual services. Thus, the classes that provide the server's functionality can not be expected to be ubiquitous. Rather, they are tailored to the server's demands. Therefore a way must be found to load classes as they are needed. In Java, classes that are referenced in an applet are loaded through the network from the applet's home site. Applying this to the mobile agent approach means at least prolonged on-line sessions and successive connection attempts. This evidently violates some of the basic mobile agent requirements. Other problems emerging in this context will be explained in the next section. A better solution is to support the packaging of a set of classes into a common transport vehicle. Here, the OSM *service profile* appears as an appropriate persistency mechanism that provides sufficient flexibility to embed Java classes as well as control flow objects.

Additionally, the following characteristics lead to the choice of Java as the language environment for the mobile agent system implementation:

- ubiquity of the Java virtual machine and thus supporting environments,
- close integration into the World Wide Web infrastructure,
- sophisticated and standardized supporting libraries.

The following section discusses how mobile agents are represented in OSM, how migration is effected, and how agents interact with one another. Finally, the overall agent architecture is sketched.

3 The OSM mobile agent architecture

This section presents the fundamental ideas of a Java-based agent system that has been devised in the scope the OSM project. Since the profile mechanism as a persistent *and* movable storage plays the key role in the implementation a closer look is taken at this technique first.

3.1 Service profile

The original requirement to the service profile was the ability to describe any services that can be offered in the electronic market. This comprises the server interface, server representation to the user (GUI elements) and eventually method invocation sequences allowed. Since the profile is transferred to and processed at the user's local computer, it must be possible to serialize all profile elements. This means that the service profile in its current state can also be saved on the user's disk. The session with the server can therefore be resumed later, probably on another host.

Since it can not be foreseen what elements are needed to describe a specified service offer, the profile must be general enough to allow new types to be defined. This implies that some meta information must be included in profile as well: for each element it must be possible to dynamically determine its type and its name. For this, the profile is structured as follows (see Fig. 1):

- There are three levels that make up the profile:
    1. The meta information that describes standardized type constructors. Examples are OsmInteger, OsmRecord, OsmArray and s.o. These are also called type objects.
    2. The meta information about concrete types e.g. ClassRecord — a record with two fields: name of type String and ClassDefinition of type Opaque. The concrete types are created with the help of the type objects. Thus they are called object types.
    3. The actual information. This is constituted from the instances of the object types. For example, there can be an instance „Agent“ of the type ClassRecord.
- Every instance has a reference to its object type and vice versa. This enables dynamic inspection of the profile contents. Also, it promotes the standardization of some elements other OSM components expect to find in the profile.
- A user is allowed to change the profile content in an individual manner. In the case of profile serialization, all changes are saved persistently or transferred to another location within the profile.

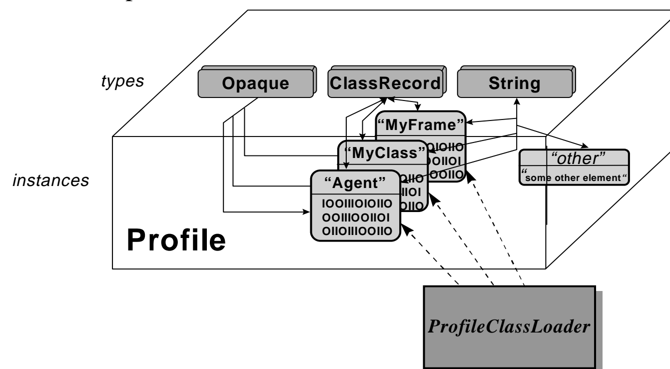

Fig. 1: Internal profile structure

It must be noted that a control description can also be embedded in the profile. The form of the description may be chosen from a wide range of concepts (from a petri net assuming an evaluator, up to coordinating Java classes) in a flexible way. Thus the profile can be used not only as a passive service offer specification but also as an active component in a workflow or mobile agent system as will be described in the rest of this section.

3.2 Standardization level

The introduction claimed that the usage of the agent system should not require any special configurations but only ubiquitously available technology (Java virtual machines are considered as such. Manipulation of the virtual machine to implement an agent system is not a choice in this paper). However, this cannot be fulfilled in its entirety since it requires at least some standardized conventions. For the sake of flexibility and autonomy it is desirable to minimize the level of such a standardization. Especially the classes constituting an agent should not be standardized, since this would restrict the personalization of an agent and may entail some operational constraints on its functionality. This implies that the agent must carry some meta information about classes it uses with it. Although it is not advisable to standardize classes themselves it is still possible to agree on a name of a main class that is used to create the first object of the agent without a loss of operational autonomy. Names of other classes whose objects may constitute the agent should not be standardized since it can not be predicted how many classes can be involved and what semantics is associated with them.

3.3 Embedding classes

What does this mean for the packaging of classes during agent migration? The suggested general execution sequence for (re-)building an agent is as follows. At the beginning, a class with a standardized name, say „Agent“, is loaded, and a new instance of this class is instantiated. This instance receives also a reference to the profile so it can build up itself up to the state it had before migration using other profile elements. Then the instance's execution is started as a Java thread and continues until a „move“ command is to be performed next, transferring the profile to the location specified as a parameter of the „move“.

As a first idea, it appears reasonable to store Java „Agent“ class bytecode in the profile and to let a Java class loader search for the class in the profile. For it, it is possible to take advantage of the fact that all profile elements are named. This concerns both meta information and instances. Assuming there is a standardized type „ClassRecord“ which describes a record with at least one field called „ClassDefinition“, a „ProfileClassLoader“ can search for an instance called „Agent“. The opaque „ClassDefinition“ contains the bytecode of the class that will be used to instantiate the agent's main object as a first step to revive it.

It is not difficult to generalize this algorithm: every class (say „MyClass“) which is used in „Agent“ has to be stored as an instance „MyClass“ of the type „ClassRecord“. This is continued recursively for other classes (e.g. classes used in „MyClass“). The

following code illustrates this algorithm as it is currently found in the implementation of the profile class loader:

```
public Class LoadClass( String sClassName ) {
...
OsmType     ot = profile.getType("ClassRecord");
Enumeration en = ot.enumOverInst();
boolean     proceed = true;
OsmInstance oi = null;

while( proceed )
  if( en.hasMoreElements() ) {
    oi = (OsmInstance)en.nextElement();
    // the instance must have the same name as the class
    // e.g. sClassName="Agent"
    if( ( oi.getName() ).equals( sClassName ) )
      proceed = false;
    else
      oi = null;
  } else
      proceed = false;
if( oi == null ) {
  throw new ClassNotFoundException("Could not find class" +
sClassName);
}
// get the array of bytes that is the class's byte code
OsmInstance oioClass = (OsmInstance) (
  (OsmRecordInstance)oi).getElement("ClassDefinition");
byte barClass[] = (byte [])oioClass.getValue();
// define the class
Class clDefinition = defineClass( barClass, 0, barClass.length);
return clDefinition;
...
}
```

There are several advantages of using this technique:

1. Only the „Agent“ class must be mentioned explicitly while searching for classes. All other classes are found using the common Java mechanism: the class loader of the object's class is asked if some type is unknown. This means, the programmer does not have to worry about class loading provided all classes are stored in profile under consistent names.
2. Class name conflicts are prevented: suppose, there are two agents that both have objects of classes with the same name although the classes are different (say „MyFrame“). Since the Java loading mechanism uses a different loader for each class applying the wrong class is avoided. Some approaches to implement Java-based agent systems suggest to find out whether all necessary classes are present at the remote host before the actual agent is transferred [Kov96]. The question here is how to cope with the situation where a wrong class with the correct name is found at the destination. Additionally, this approach requires prolonged online communication, a fact which contradicts to the agent-oriented paradigms.

Using this profile class loader concept, the general scenario can be refined as follows. The profile class loader returns the „Agent“ class and the system creates an instance of the class. The result of the operation is casted to a predefined interface (say

„BaseAgentInterface“) which is indeed a part of a fixed convention. Note however, that the agent code can still be any class that implements this Java interface. The interface provides a method to revive the agent. This means, all information that is necessary for the agent to restore its old state and to continue execution is passed to it. This mechanism is described in more detail in the next two subsections.

3.4 Agent migration

The problem of state recovering occurs only in the context of agent migration. Before an agent is transferred to another location it must be converted into a form that

- is platform independent
- preserves the current agent state
- preserves the agent's integrity

The profile mechanism fulfills all these requirements. Thus, the following agent migration procedure is supported. During its execution the agent writes all information that it will need at other locations in its profile. This information will be available after the transmission has taken place. Since the agent writes the data itself, it knows where it will find it within the profile afterwards. After the profile has successfully arrived at the specified new location the local (old) agent thread is terminated. The further execution will take place at the destination host.

All components involved in agent processing are listed below (see also Fig. 2). These actually constitute the current OSM agent system.

- The *Engine*: receives agents, manages them and allows for their communication. The class Engine must provide a functionality defined by a „BaseEngineInterface“ which is subject to standardization.
- The *Assembler*: created by the engine on the agent arrival, sets up the agent for execution.
- The *ProfileClassLoader*: created by the assembler, locates and loads agent’s classes.
- The *Dispatcher*: created by the engine, is responsible for agent splitting.
- The *Forwarder*: created by the engine or dispatcher in the case the agent wants to move.
- The *Synchronizer*: created by the engine, responsible for agent synchronization.

Compared to the „all-in-one-engine“ approach, this structuring avoids bottlenecks and allows for higher autonomy and security as shown below. The following, more detailed discussion, aims at making the cooperation of these components clearer.

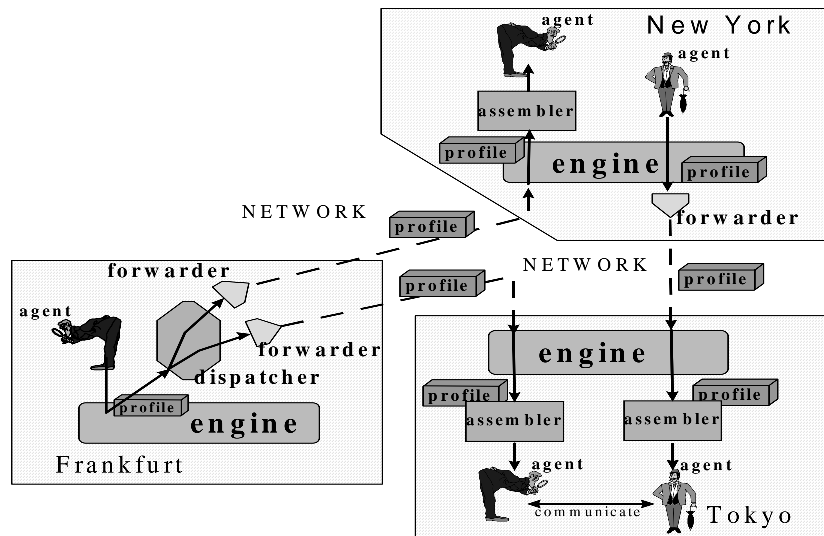

Fig. 2: Agent System Components

Bootstrapping

On indication of an incoming agent, the engine creates an assembler object that actually receives the agent and sets it up. The information passed to the agent by the assembler includes the reference to the engine and to the profile. The corresponding agent method is called „`reviveAgent`“ and is part of the „`BaseAgentInterface`“. This method is overloaded in a manner that the agent fetches the information which is relevant to its state from the profile. If the agent needs some information from the environment (at the new location) it contacts the engine. The agent presumes that the engine provide certain functionality to access the environment. Due to this indirection, a better level of security is achieved since the locally installed engine can be specified, tested and controlled by the local service provider — who should trust it then without being afraid of direct manipulation of his environment by incoming agents. At the end of the „`reviveAgent`“ method, the agent starts its execution as a Java thread. A „`run`“ method determines the next activity to be performed by the agent. The local activity continues until a „`move`“ method is invoked. Another way to terminate execution is to call the „`destroy`“ method being part of the „`BaseAgentInterface`“. All resources associated with the agent and managed by the engine are freed then and the agent's thread is destroyed.

The next paragraphs describe other methods provided by „`BaseAgentInterface`“. These constitute the fundamentals for inter-agent concurrency and communication approaches.

Concurrent agent inter-working

It was claimed above that electronic markets present the domains where agents can be employed as „value-adders“ by simplifying service access or combining more than one service automatically within a single task given to them. Additionally, electronic

markets are constituted from naturally concurrent business processes which correspond well to inter-agent concurrency.

In the OSM agent system, inter-agent concurrency is implemented based on the „agent splitting“ approach. Splitting here means that an agent is cloned several times. However, local variables of each instance can be overwritten before or after cloning, so they are not necessarily equal. Thus, depending on these variables, the further execution may be carried out differently. All instances created differ at least in their identifiers. Also, their destinations can be distinct. These can influence the decision about the next activity to be performed after transport. If the agent does not supply any names, the instances are named uniquely by the engine.

A „`split`“ method passes all necessary information to the engine that actually initiates the splitting process: The engine creates a *dispatcher* object which manages the further splitting. For the actual transfer, a *forwarder* is employed. Figure 2 illustrates this sequence. The receiving engine does not distinguish between the original agent and the clones.

Often, some or all of the created subagents need to be synchronized in order to exchange information collected. For it, it is necessary to make an agent wait for another one. This can be another subagent created by an earlier split or it can be a completely different agent. In any case, the agent must be able to identify the other agent it is waiting for. In the OSM agent system, the agent's unique identifier is currently used for this. Since it may not be desirable that the agent is able to access the information about what agents are present at the local host, the synchronization is carried out indirectly by a *synchronizer* object. For this, the agent invokes a „`synchronizeAgents`“ method being part of the „`BaseEngine-Interface`“ again. Then it suspends itself.

The engine creates a synchronizer object and passes to it information about which agent is waiting, which agent is being expected to arrive and information about all agents at this location. If agents being expected have not arrived yet, the synchronizer periodically fetches information about present agents from the engine. As soon as the agent awaited arrives or after a specified timeout the synchronizer wakes up the waiting agent. It is anticipated that the agents synchronize themselves in order to exchange some data. This means, after the synchronization, agents will probably set up communication with each other.

Inter-Agent Communication

Since it can not be predicted what information should be exchanged by the agents, it is desirable to develop a very generic communication technique. A first idea is to require a reference to the communication partner from the engine and then to cast the reference from the „BaseAgentInterface“ to the actual agent type. This approach supposes that the communication entities are informed about the implementation details of each other. This is not too unrealistic, especially in the case of formerly splitted agents. However, the current serialization package implementation does not allow such references to be casted to the agent's actual type although the types are equal. Thus, in the current agent system implementation, a less elegant but generic approach is utilized. A method „`communicate`“ (with a parameter of type „`Object`“) is provided by the

„BaseAgentInterface“. It should be noted, that the agent communication does not involve third instances like the engine. Therefore it can be kept confidential and be considered as a secure channel. As a part of the given object parameter, credentials could be passed that authorize and authenticate one agent to the other.

This subsection closes the discussion about the OSM agent system functionality. As it is stated above, agents present a technique that seamlessly fit into electronic market technologies. This is illustrated by the example in the next section showing where the other concepts of the OSM environment may support the use of agent.

4 An information retrieval scenario

For now, the usual way to find some information, e.g., about stone-age wooden-made tools is by asking a user's favorite search engine. The OSM infrastructure could instead offer a *catalogue* of specialized agents. The choice of the agent that seems to be most appropriate is done with the *generic client* which may also be used to parameterize the agent by the contents of the user's request. The agent's own program analyses the request and determines that at least two different areas of interest are involved. Therefore it splits itself and one of the clones migrates to a site containing a historical database in New York and the other visits a technical library in Tokyo (see also Figure 2). After arrival, these agents contact the sites' local resources and retrieve information they are interested in. This process also involves the OSM payment supporting services since the information resources might be offered commercially. This in turn includes the OSM security features to keep the information delivered confidential. Since the historical database is a rather large one, the first agent may like to send some intermediate results to Tokyo where the information is merged with those retrieved by the second agent. This transfer could be done by a different agent (say an *information messenger*). Finally the Tokyo agent goes back to the user's home at Frankfurt and reports its result to the user.

5 Conclusion and outlook

This paper gave a view of the motivation for Java-based mobile agent systems and some important implementation aspects. It was shown how agent persistence, migration, synchronization, and communication have been integrated into an electronic market architecture.

The most important features of the OSM mobile agent system can be summarized as follows. A minimal standardization level is given that does not hinder the customer in highly individualized application development. Also class naming autonomy is guaranteed for agents. By encapsulating its own classes, the agent is able to warrant a certain level of security.

This contribution emphasized the question of agent persistency and interaction. Further aspects, such as the authentication infrastructure, payment aspects, or agent-based negotiation support are addressed by the OSM architecture, but not discussed in detail. On the other hand, the question of agent programming tools still remains open: In the current version, an agent is programmed manually — including design and manipula-

tion of the control flow. One of the next tasks is thus to ease the work of the programmer on control flow specification by applying special software tools. These tools will then be a part of the overall OSM agent development environment.

6 References

[CFF+92] H. Chalupsky, T. Finin, R. Fritzson, D. McKay, S. Shapiro, G. Wiederhold: "An overview of KQML: A knowledge query and manipulation language". Technical Report, April 1992

[CGH+95] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Paris, G. Tsudik: "Itinerant Agents for Mobile Computing". IBM Research Report RC 20010

[Jor96] Jordan, M.: „Early Experiences with Persistent Java“. In Proc. of the first International Workshop on Persistence and Java, University of Glasgow, 1996

[Kov96] E. Kovács: Advanced Trading Service Through Mobile Agents. In: *Proc. Trends in Distributed Systems ‘96*, Aachen 1996, pp. 112-124

[MRK96] T. Magedanz, K. Rothermel, S. Kruse: "Intelligent Agents: an Emerging Technology for Next Generation Telecommunications?". In: Proc. IEEE INFOCOM, San Francisco, USA, März 1996

[Mer96] M. Merz: "Elektronische Dienstemärkte - Modelle und Mechanismen zur Unterstützung von Handelstransaktionen in offen verteilten Systemen". Diss., Universität Hamburg, November 1996

[MML94] M. Merz, K. Müller-Jones, W. Lamersdorf: „Service Trading and Mediation in Distributed Computing Systems". In: L. Svobodova, Ed., Proc. 14th 'International Conference on Distributed Computing Systems', Poznan, Poland, IEEE Computer Society Press, 1994, S. 450-457

[MML96] M. Merz, K. Müller-Jones, W. Lamersdorf: „Agents, services, and electronic markets — how do they integrate?“. In: A. Schill, O. Spaniol, Ed., *Proc. International Conference on Distributed Platforms ICDP ‘96*, Feb. 1996

[MTL96] M. Merz, T. Tu, W. Lamersdorf: „Dynamic Support Service Selection for Business Transactions in Electronic Service Markets". In: *Proc. Intl. Workhop on Trends in Distributed Systems*, Aachen 1996, Springer, Berlin, Heidelberg New York 1996, pp. 183-195

[Schm93] B. Schmid: "Electronic Markets". In: *Wirtschaftsinformatik*, 35 (1993) 5, S. 465-480.

[Whi94] J.E. White: „Telescript Technology: The Foundation for the Electronic Marketplace“. White Paper, General Magic, Inc., 1994