Universität Stuttgart
Fakultät Informatik

Efficient Code Migration for Modular Mobile Agents

Fritz Hohl, Peter Klar, Joachim Baumann

Email: {`Fritz.Hohl,Joachim.Baumann}@informatik.uni-stuttgart.de`

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

**Efficient Code Migration for
Modular Mobile Agents**

Fritz Hohl, Peter Klar, Joachim Baumann

Bericht Nr. 1997/06
September 1997

Efficient Code Migration for Modular Mobile Agents[1]

Fritz Hohl, Peter Klar, Joachim Baumann

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

`{Fritz.Hohl,Joachim.Baumann}@informatik.uni-stuttgart.de`

Abstract

This paper examines how an efficient code migration component for mobile agents can be designed, an aspect crucial to the migration performance and thus, for the overall performance of such a system. Therefore, after listing the requirements of code migration in this area, and a description of the code migration mechanisms of the Java applet model, a more appropriate approach is proposed. This new approach uses two mechanisms to transport code: one that is able to get classes in an efficient manner and one that is able to get any valid class by its reference. Both mechanisms use a code replacement policy that is able to handle a space restricted class storage. Finally the approach uses digital signatures to protect the code migration component, the *codeserver*, against modification attacks.

1 Introduction

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

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

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

The paper is organized as follows: Section 2 lists the requirements of code migration in mobile agent systems. Section 3 describes the code migration model of the Java applet system, which is widely used in Java-enabled WWW browsers. In Section 4, a model of code migration is presented that fits the requirements found in section 2 better. Section 5 discusses possible architectures of a code migration component, the codeserver. The paper closes with a conclusion and remarks about future work.

2 Requirements of code migration in mobile agent systems

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

there is a code source for any given class

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

code can be loaded at least when:

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

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

- an agent wants to load a new class explicitly.

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

i.e. single points of failures must not occur.

class references designate code versions known to the programmer

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

code is protected against modification attacks like viruses and trojan horses

when code is not transported in one piece, but contains a list of class references that are used to find the correspondent classes, attacks can use modified versions of a class e.g. by inserting “intrusion code”.

code is migrated in an efficient manner

since code migration represents an essential part of the overall performance of the agent migration, the code migration component has to perform well in order to allow the mobile agent system to perform well.

the code migration mechanism fits into the asynchronous processing model that is offered by mobile agents

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

the code migration mechanism fits into the organizational model of the mobile agent system

e.g. if we assume an organizational model that employs an ‘egocentric’ approach of the participating institutions, we cannot successfully use a code migration mechanism that relies on a “selfless acting” mobile agent infrastructure.

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

3 Related Work

The problem of code transport is common not only to mobile agent systems, but also to some of the mobile object and to mobile code systems. They can be divided, according to their code requesting method, in two groups:

1. systems that view code as one piece of code and which ship it as a whole
2. systems that use the Java applet code requesting method

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

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

3.1 The Java Applet Code Model

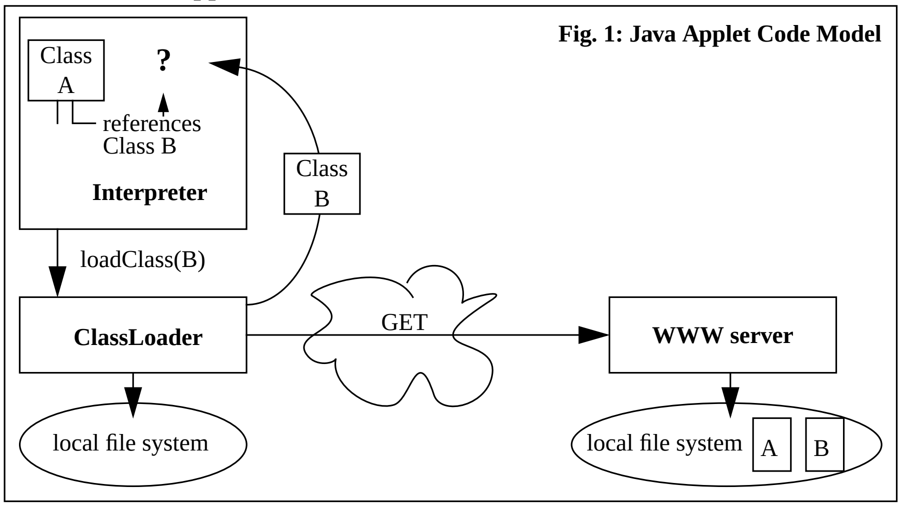

Fig. 1: Java Applet Code Model

Fig. 1 presents the mechanisms that are used to get new code into a Java-enabled browser: In the code of a class A, an object of class B has to be created. Since class B is not found in the set of loaded classes, the java interpreter asks a system component, the *ClassLoader* for this class. Depending on the source of class A, the ClassLoader tries to get the code for class B from the same source. When A has been loaded from the local file system, the ClassLoader is looking there for B, when A has been loaded via HTTP from a WWW server, it tries to get class B from the same WWW server using the same URL prefix. Once the ClassLoader gets the class file, it initializes class B, which can then be used by the interpreter to create an instance of it.

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

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

4 The Codeserver Model

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

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

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

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

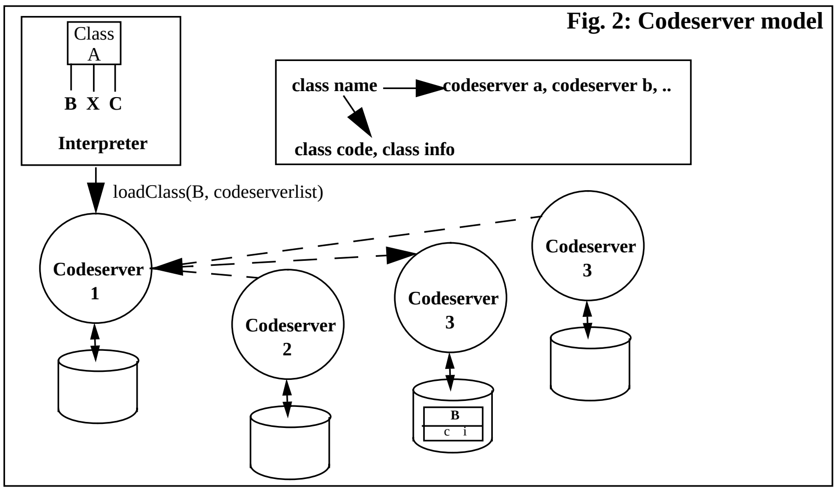

Fig. 2: Codeserver model

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

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

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

5 Codeserver Architectures

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

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

5.1 The basic mechanism

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

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

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

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

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

5.2 The standard mechanism

The purpose of the standard mechanism is to retrieve new classes as *fast* as possible. This is a problem that has until now not been examined in the area of mobile object systems. Therefore, new mechanisms suited to that specific area have to be developed. Our approach tries to improve efficiency by fetching code fragments not necessarily from the source of the migrating instance, but from other, “neighboring” codeservers. The neighbor relationship is expressed through the codeservers’ distance (the distance being a function of delay and bandwidth). Therefore, two things have to be computed: the set of possibly neighboring codeservers, and the real “distance” to them both in terms of roundtrip delay and transmission speed.

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

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

In order to further speed up the mechanism, neighbors exchange lists of stored classes, so a codeserver does not have to first ask around for a class. As soon as these lists are mutually known, only updates of the lists instead of whole lists are exchanged.

An additional way to make the code transport faster is to prefetch classes that may be used in the future. For this purpose, a class provides additional information about which classes have been loaded how often by requests coming from this “main”-class in the past. The codeserver then can get at least the most frequent classes in advance. If the behavior in the past allows to foresee the future class references, this mechanism allows to load classes that not even occur in the code of the main class. This means also, that the loading policy is not on-demand, but greedy in a way that the m most frequent referenced classes are loaded while the main class is executed. This and other mechanisms that try to exploit informations about the (potential) behavior of interpreters allows the interpreter to use exactly one dedicated codeserver, thus optimizing network communication. If the normal operation would allow the interpreter to use more than one codeserver, several codeserver instances would try to prefetch code in parallel, which is inefficient if done without synchronization. Further optimization could exploit the fact, that it is more efficient to get classes in a bulk instead of one by one from a partner, since the overhead to establish a connection would grow higher if for each one class a connection has to be made.

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

5.3 Code transport

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

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

Interpreter - Codeserver

`Class getClass(Classname)`
This method is used by the interpreter to request a class from the codeserver by its name.

`Address[] getCommunicationPartners()`
The codeservers uses this method to get the known communication partners (i.e. interpreters) from the interpreters it provides with code.

`Address getCodeserver()`
This method is used by codeservers to request the name of the (mainly) used codeserver of this interpreter. This method can be used to find new neighbor candidates.

Codeserver - Codeserver

`Class getClass(Classname)`
This method is used to request a class by its name.

`Classname[] getDirectory()`
This method is used to request the entire list of classes stored in a codeserver.

`Classname[] getDirectoryDelta()`
This method is used to get an update of the list of classes stored in or deleted from a codeserver with relation to the last directory request.

Codeserver - Programmer

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

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

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

5.4 Code replacement policies

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

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

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

What we have to consider is that prefetched classes (compare Section 5.2) that are not already requested by an interpreter, will probably be needed in the near future. Therefore the replacing algorithm should give these classes more time before they will be replaced (with the same argument as above, the interpreters should inform the codeservers about classes that have been garbage collected).

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

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

5.5 Security aspects

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

6 Conclusions and future work

In this article we presented a code migrating mechanism that fulfills the requirements of mobile agent systems:

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

The described mechanism has been currently implemented (see [Kla97] for details) in our mobile agent system, Mole [SBH97, BHRS97], which is built on the Java language environment [GJS96]. We expect a significant performance increase of the agent migration by using a codeserver, and we will, therefore, compare this potential increase by measuring the migration times in agent applications such as active documents or network management. Other positive effects of using such a code migration component is the reduction of the size of an agent system node, which allows the usage of the system even in smaller devices, and the solution of some system problems such as versioning of classes and the protection of code. Finally, we will try to find further optimization mechanisms like compression of classes by examining code request profiles of different applications.

Literature

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

[BHRS97] Baumann, Joachim; Hohl, Fritz; Rothermel, Kurt; Straßer, Markus: Mole - Concepts of a Mobile Agent System, Technical Report No. 1997/15, Faculty of Computer Science, University of Stuttgart, Germany, 1997

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

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

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

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

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

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

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

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

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

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

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

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

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