Attacking the Migration Bottleneck of Mobile Agents


Peter Braun, Swinburne University of Technology
Ingo Mueller, Swinburne University of Technology
Ryszard Kowalczyk, Swinburne University of Technology
Steffen Kern, Friedrich-Schiller-University Jena, Germany
19 January 2005
Table of Contents

Abstract ................................................................................................. 1

Introduction .......................................................................................... 1

Background: Migration Strategies .......................................................... 3

Adaptive Transmission of Code ............................................................... 5
  Programmable Migration Strategies ....................................................... 5
  Decision between Push and Pull ............................................................. 5

Unnecessary Class Downloading ............................................................. 6

Code Cache ............................................................................................. 6

Class Splitting ........................................................................................ 8

State Transmission .................................................................................. 10

Conclusion and Outlook .......................................................................... 11

References .............................................................................................. 11
Attacking the Migration Bottleneck of Mobile Agents

Abstract
Mobile agents were introduced as a new design paradigm for distributed systems to reduce network traffic as compared to client-server based approaches simply by moving code close to the data instead of moving large amount of data to the client. Although this thesis has been proved in many application scenarios, it was also shown that the performance of mobile agents suffers from too simple migration strategies in many other scenarios. In this paper we identify several reasons for mobile agents' poor performance, most of them are related to the Java programming language. We propose solutions to all of these problems and present results of first experiments to show the effectiveness of our approaches.

Keywords
Mobile Agents, Code Migration, Migration Optimization

Introduction
Mobile agents have been introduced as a new design paradigm for distributed applications by James White [1]. A mobile agent is a program that can migrate from host to host in a network of heterogeneous computer systems and fulfill a task specified by its owner. It works autonomously and communicates with other agents and host systems. During the self-initiated migration, the agent carries its code and some kind of execution state with it. What comprises the execution state depends on the underlying programming language and is, in the case of Java for example, the serialized agent (which is an object of a specific class) and does not contain information about the state of the Java virtual machine. On each host they visit, mobile agents need a special software that we name agency, which is responsible to execute agents, provides a safe execution environment, and offers several services for agents residing on this host. A mobile agent system is the set of all agencies of the same type together with the agents running on these agencies as part of an agent-based application. To refer to a specific project or product, for example Aglets [2] or Grasshopper [3], we use the notion agent toolkit.

In this paper we solely focus on the problem of performance of mobile agents. One major argument in favor of mobile agents is code-shipping versus data-shipping. In a client-server based application, a single remote procedure call (request) might result in a huge amount of data sent back to the client (reply) due to the lack of precision in the request. Instead of transferring data to the client where it will be processed and filtered and might cause a new request (data-shipping), this code can be transferred to the location of the data (code-shipping) by means of mobile agents. In the latter case, only the relevant data, i.e. the results after processing and filtering are sent back to the client. This advantage was already mentioned in the original paper by White [1] and later discussed by Chess et al. [4]. This advantage has been scrutinized in the last years by many different research groups for different application domains and it has been verified in general [5–7].
However, these papers also pointed to some situations, in which mobile agents performed worse than the client-server approach. It is obvious to see that the code-shipping versus data-shipping argument is only valid if, simply speaking, the mobile agent’s code that has to be transmitted is not larger than the amount of data that can be saved by the use of a mobile agent. It depends on the size of the request, the size of the reply, the code size and some other parameters to decide whether mobile agents perform better than client-server approaches. In the following, we will call this problem the migration decision problem. Picco [8] proposes a static design decision between client-server and mobile agents, based on a mathematical network model of the application and its environment, and using estimations of the previously mentioned parameters. This approach is limited because it does not consider the possibility of a mixture of agent migrations and client-server communications, which has been theoretically proven to perform best under the assumption of full knowledge of the environment [9]. Since even the latter approach is based on estimations of the migration parameters, this decision might be wrong, if parameters of the mathematical model turned out to be different in practice.

As part of our project, we are working on techniques to solve the migration decision problem solely by mobile agents. We do not propose a hybrid technique, where it is (statically or dynamically) decided between remote procedure calls and code migration, but we suggest using mobile agents in any case. The idea is to replace remote procedure calls by agent communication and to use agent migration to optimize the performance of the application, if possible.

The first step in this project was to identify and understand why mobile agents sometimes perform worse than client-server approaches. This paper presents our experiences and lessons learned from analyzing the migration process of existing Java-based mobile agent toolkits. We have identified several reasons for mobile agents’ poor performance, which can be summarized using the idea of migration efficiency. The migration efficiency of a mobile agent defines how many code units (on the level of statements, methods, or classes) and data units (variables or objects) of an agent have been used (read or written) on remote agencies proportional to the number of code units and data units that have been transmitted. An agent has a low migration efficiency, if many code units or data units have been transmitted superfluously, i.e. they have not been used at remote agencies. Note that a migration efficiency greater than 1 can make sense, since agents’ code can be deployed or cached in advance. It should be clear that the performance of a mobile agent is directly influenced by the migration efficiency. The specific contributions of this paper are:

- We describe common migration techniques used in today’s Java-based mobile agent toolkits and identify their limitations.
- Client Characteristics We have developed a new mobility model, named Kalong, which contains solutions for all identified problems.
- We present results of first experiments to show the effectiveness of our approaches.

The rest of this paper is structured as follows: In Sec. 2 we give an overview of the current state-of-the-art technology in the area of mobile agent migration techniques. In the following sections 4 to 6, we identify shortcomings of the Java programming language and its built-in class downloading technique and present solutions for all these problems. After having discussed code migration issues, we discuss in Sec. 7 the problem of state migration and present our approach to optimize
the migration efficiency for state transmission. Finally, Sec. 8 concludes the paper and gives an outlook to further development.

Background: Migration Strategies

In the following, we restrict ourselves to Java-based mobile agent toolkits, because the Java programming language has become the de-facto standard for programming of mobile agents; almost all new toolkits developed in the last six years are at least partially programmed in Java. The reasons for this are many built-in functions, for example object serialization, dynamic class loading, and the sand-box security mechanism, which simplify the development of a mobile agent toolkit. For some time, people suppressed problems caused by the Java programming language. Only recently, Binder [10] points to resource monitoring problems and Roth [11] identifies several security holes making it unlikely to ever completely protect agencies from malicious agents. In this paper, we align ourselves with these arguments while focusing on code and data relocation techniques.

When a mobile agent decides to migrate to another host, the underlying agency is responsible to stop agent execution, serialize the agent, and to transfer the agent’s state to the destination agency. The destination agency receives and deserializes the agent’s state and then re-starts the agent by creating a new thread and invoking a specific method. (Here, we are not interested in the discussion whether mobile agent toolkits should provide weak or strong migration.) This framework can be considered as the least common denominator of all mobile agent toolkits. It does not describe how code transmission works, because this can be implemented in different ways. In the following, we use the term migration strategy to describe how code and data are relocated. We can identify roughly two classes of migration strategies called pull and push strategies. The first strategy uses a built-in Java technique to download classes on demand. If some classes are needed during execution but not yet available locally, the Java class loader tries to find these classes remotely, for example at the agent’s home agency (the one on which the agent was started and which contains all the code). We call this a pull strategy. This strategy class can be further divided into pull-per-class and pull-all-classes. The first strategy opens a new network connection for every class, whereas the second strategy loads all class files as one package (e.g. a JAR file), if any of its classes must be loaded. Pull strategies are used for example in Mole [12], Grasshopper [3], and Jade [13]. The intuitive advantage of the pull-per-class strategy is that only necessary classes are loaded from the home agency. Thus, we can expect a high migration efficiency for code. We will see later that our intuition of necessity does not match the Java language definition. A major drawback of all pull-oriented migration strategies is that there must be an open network connection to the home agency, which makes this strategy inappropriate for agents that have been started on a node with unreliable or volatile network connections, for example a mobile device. To cope with these situations, Grasshopper enables agents to load classes from the last agency they visited. In Mole it is possible to manually deploy agent’s code to code servers other than the home agency (has not to be a complete agency, may be only a Web server) in advance.

The second class of migration strategies is named push strategies. The code of an agent (together with the code of all referenced objects—the class closure) and the serialized agent state (the object closure), are transmitted at once. This strategy corresponds to one important characteristics of mobile agents, that is autonomy: The agent does not need a connection to its home agency. At a first look we could consider this strategy to be fast, because only one transmission is necessary for the complete agent. However, a major drawback is that some code is transmitted to the destination
site that is probably never used—the migration efficiency might be poor in this case. In addition, it is not trivial to determine the class closure in Java, since the Java API only provides methods to request the type of attributes, method parameters, and return values—but not local variables.

It is not surprisingly that neither pure push nor pure pull strategies lead to the optimal network load and application performance in all cases. A simple comparison of the push and the pull-per-class strategy for an agent’s tour to 7 agencies in a high-bandwidth (100 Mbit/sec) local area network shows (see Fig. 1) that if only a few classes are needed for execution at all agencies, the pull strategy performs better. In contrast, if many classes are needed, the push strategy would have been the better choice. At this point, we cannot be more precise than only say few and many, because the number of classes at which both strategies produce the same network load depends on many parameters.

![Figure 1. Round trip time for an agent to 7 agencies using different migration strategies.](image)

Our overall goal is to increase the migration efficiency, that is, to develop techniques so that we only transfer code and data that are needed on remote agencies with high probability. In this paper, we do not consider other technique to optimize the migration process, for example to reduce the number of bytes by compressing classes before transmission [14] or using other intermediate code representations than Java byte code [15]. Those techniques can be seen orthogonally to the approaches we outline in this paper. The following list summarizes all approaches that we present in this paper:

- Using adaptive transmission of code and data it is possible for the agent to decide during runtime which classes should be pushed to the next destination and which classes should be pulled at the destination agency later. The agent can decide which data items must be transferred to the next destination and which must be sent back to the agent's home agency.
- Even, if we go for a pull strategy, the Java class loading mechanism might load classes that are never used, which decreases the migration efficiency. We give an example to illustrate this problem and present a straightforward fully-automatically solution to prevent unnecessary class downloading.
Code caching can be a powerful technique to prevent code transmission and therefore to increase the migration efficiency. Due to security restrictions and problems in distinguishing different versions of the same class, many toolkits disable the built-in Java code cache. We present an alternative approach for code caching that goes beyond Java’s capabilities and increases code migration efficiency significantly.

Finally, we propose to transmit code on the level of single methods rather than on complete class files. We learned that different methods of the same class might have various execution probabilities. Therefore, it makes sense to split classes into smaller transmission units.

All these approaches have been implemented in our new mobility model, Kalong, which is also available as independent software component. Kalong is designed to be adaptable to all Java-based mobile agent toolkits. We are currently working on the integration of Kalong into Jade, and Semoa [16].

Adaptive Transmission of Code

Our thesis is that mobile agents should be able to adapt their migration strategy according to specific environmental parameters, as for example, the code size of each class, the probability that a class is used at the next destination, network bandwidth and latency, etc. This could result in a mixture of push and pull strategies, for example, to only push the agent’s main class and to dynamically load other classes. In none of the available toolkits the agent (or agent programmer) can influence the migration strategy during runtime in order to select which classes shall be pushed or pulled.

Programmable Migration Strategies

We propose programmable migration strategies. Imagine a migration strategy as a program for a virtual machine that is responsible for agent migration. Basic commands of this virtual machine are, for example, sending an agent’s state or sending an agent’s code unit (a unit can be a class or a set of classes). Classes that were not sent during migration must be pulled on demand later. Additionally, the migration strategy also defines, which data items (part of the agent’s state) should be migrated to the next destination or, for example, sent back to the agent’s home agency. We come back to this issues later in Sec. 7.

It should be obvious that by using these two primitives for state and code transmission, it is possible to describe all migration strategies that we have introduced in the previous section. For example, to describe the push strategy, we define all classes to form a single code unit, which is sent along with the agent’s data to the next agency. To describe the pull-per-class strategy, we define that each class forms a single code unit and none of them is transmitted along agent’s code.

Decision between Push and Pull

The problem of deciding which classes should be pushed or pulled is subject of ongoing research. The main idea is to analyze the Java byte code of an agent before it is started to gather information about execution probability for each code unit. The analysis starts with analyzing control flow and the constructing an intra- and interprocedural control flow graph [17]. Afterwards a call graph is built whose vertices represent methods with at least one entry point and whose edges designate a possible method call. During our upcoming analysis steps each vertex will be labeled with the execution probability of the method it represents and each edge (a,b) with the probability that b is called by a.
To construct this call graph we use a class hierarchy analysis followed by a rapid type analysis to reduce the number of edges in the graph \[18,19\]. We then perform a data flow analysis \[20\] and value range propagation \[21\] to determine the execution probabilities of each branch following a conditional jump. To execute a fast value range propagation the control flow graph is transformed into static single assignment form \[22\]. For branches where value range propagation is not able to determine any probability information, several heuristics are applied \[23\]. These results are combined with some heuristics, for example to filter out classes that are already available at all agencies with a very high probability, for example all Java classes.

Finally, we use these results to select code units for pushing to the next destination by specifying a necessary probability interval of class execution. For first experimental results of mixing both techniques, we refer to Sec. 6.

### Unnecessary Class Downloading

During our investigations we have observed that our intuitive understanding of necessary classes contradicts, to some extent, Java’s language specification. We have noticed that classes were downloaded by the Java virtual machine, although they were never used by the agent. For example in Fig. 2 we have found that the Java virtual machine loads all classes of object attributes (attribute b) already during the agent’s deserialization process, even if an object attribute (and therefore its class) will never be used. The code of local variables (local variable d), parameters, and return values are as expected only loaded, if they are used. Thus, the beneficial effect of pulling code in order to reduce network load is limited to local variables—making it necessary to program mobile agents accordingly.

Fortunately, we have found a straightforward solution to this problem by changing the type of all object attributes to Serializable and performing necessary type casts when it is used (attribute c). The effect is that now classes are only downloaded, if an object of this type is used. We have implemented this code rewriting approach on the level of Java byte code.

### Code Cache

In general caching is a technique to locally store frequently used data in order to prevent expensive operations for loading this data from a distant location. In case of mobile agents, it is obvious to store agent’s code locally to avoid expensive code downloading operations. Code caching can not only improve the performance of agents that visit a specific agency twice, but also prevent code loading when two agents of the same type visit the same agency.

Java provides code caching as part of the class loading mechanism. A class loader is an object responsible for loading one or many classes; every class is loaded by exactly one class loader object. Inside a class loader, classes are identified by their name, i.e. two classes bearing the same name are expected to be identical. Different class loaders must be used to create different name spaces, which has the consequence that classes loaded by different class loaders cannot see each other.

In contrast to the general idea of class caching, the existing Java mechanism has two shortcomings. The first one is that the Java class cache only prevents code downloading (pulling)—but not code pushing. The second one is that the designer of the agent toolkit has to decide between two
alternatives regarding code cache. Either, the code cache prevents class loading even for different agents of the same type or the cache can distinguish between different versions of classes, even if they have the same name. In the first case, for example used in Jade, all agents are loaded by the same class loader. No class is loaded twice in Jade, which has the consequence that Jade cannot distinguish between different versions of the same class. In the second case, each agent is loaded by a different class loader; this is, for example, used in Grasshopper. Downloading the same class cannot be prevented, if the class is used by different agents.

```java
public class A implements Serializable {
    private B b;
    private Serializable c; // former class C

    public void run() {
        D d = new D();
        System.out.println(((C)c).toString());
    }
}
```

**Figure 2. Source code example to illustrate Java class loading.**

As part of the Kalong mobility model, we have implemented a code cache that solves both problems mentioned previously. During the migration process it is checked whether code is already available at the destination agency, without sending the whole code (compare Fig. 3). We use MD5 digests to compare two classes unambiguously. Classes are not cached as part of a class loader object, but in a separate component of the Kalong component. Thus, all class loader objects can benefit from the single code cache.

**Figure 3. UML sequence diagram to illustrate the class cache. Only the first three protocol steps are shown.**
Fig. 4 shows the result of an experiment, where an agent (of various code size) processed an itinerary twice. On the first tour, all code must be transmitted to each agency, whereas on the second tour, no code was transmitted, because it was already cached. The diagram shows the significant effect of code caching on processing times.

![Figure 4. Difference in migration times with and without using class cache.](image)

**Class Splitting**

All migration techniques and analyzes we have described so far, transfer code either on the level of classes or class packages (JAR files). We propose to transmit code on the level of methods rather on the level of classes. In this section, we first motivate our proposal by an example and then outline the class splitting process that we have developed.

```java
public class Agent implements Serializable {
    public void run() { ... } // main entry
    private void startAgent() { ... }
}
```

*Figure 5. Source code of the original class (before splitting).*

Imaginate typical example of a mobile agent that roams the Internet searching for some kind of information in different types of databases, for example for images, movies, or XML files. The agent carries specific algorithms to analyze each document type, coded in different methods of the agent’s main class. Its migration efficiency, now measured on the level of methods rather than on the level of classes, is influenced by the number of different document types found on each agency. In the worst case, the agent’s code (one class) is loaded completely, although no document type was found at all. In general, we have learned from our experience in programming small to medium mobile agent-based applications that methods of one class are unlikely to have similar execution probabilities. By taking a closer look at an agent's typical structure, we can identify four different types of methods:
- Methods only used during the startup phase on the agent's home agency.
- Methods used while the agent traverses the network.
- Methods used at the end of an agent's lifetime, which is likely to happen on the agent's home agency again.
- Methods called to handle unexpected situations, e.g. exceptions.

We can surmise that it would be beneficial to group methods from the second type into the agent's main class and load all other methods on demand. Of course, a programmer of mobile agents could implement an agent using many classes (e.g. one class per method) and by this manually reduce code granularity and increase migration efficiency. In contrast, we propose a semi-automatic approach, where the programmer defines probability ranges $P_1, P_2, \ldots, P_n$ before the agent is started. During the start process of an agent, the same code analysis as described in Sec. 3.2 is performed, which results in knowledge about execution probability on the level of methods. All methods with an execution probability element of the same range $P_i$ will be grouped together in a new class. The process of splitting classes is divided into two steps:

- Moving original methods into a new class file and ensure that they are still operational in this new environment. The new classes will be located in the same package as the original class.
- Create a stub method replacing the original method. This stub has to behave in exactly the same way as the original method, i.e. it must have the same number of arguments, same return type, and same flags. When a stub method is called, it calls its corresponding method in the split class.

```java
public class Agent implements Serializable {
    private Agent$s01 s01;

    public void run() { ... } //stub method to forward request
    private void startAgent() {
        if(s01 == null)
            s01 = new Agent$s01();
        s01.startAgent();
    }
}

//new class with home methods
class Agent$s01 implements Serializable {
    void startAgent() { ... }
}
```

**Figure 6. Source code of the split classes.**

A detailed description of this process is out of the scope of this paper. Only for illustration purposes, compare Fig. 5 and Fig. 6. The first one shows the source of the original agent and the second one shows the split agent. For sake of clarity, both figures show Java source code, although the class splitting process works on the level of Java byte code.
To evaluate the effect of class splitting, we have conducted the following performance experiment. We have implemented an agent that consists of 6 methods, i.e. one method (run) that is used on all agencies, one method (startAgent) that is only used on the home agency, and four methods to process different document types. The size of this agent was 20 KByte. After splitting this class, we obtain six new classes, each contains only one method: the class with method run has 3566 Bytes, the class with method startAgent has 1852 Bytes, and the four other classes have 3622 Bytes each. The experiment has been conducted in a local area network, in which we varied the maximum bandwidth. The agent migrates to 4 agencies and is either transferred using the push strategy or the pull strategy. In both cases, we modified the number of document types that the agent finds on each host. In case of the pull strategy, the number of document types influences the number of classes to be downloaded for this agent. The results prove our expectations, see Fig. 7. Splitting code can have a significant effect on the migration performance in small-bandwidth networks (512KBit/sec). Here, any pull strategy is never slower than the push strategy. In high-bandwidth networks (more than 1 MBit/sec) we notice that pushing all code can be faster than pulling parts of the code. This effect is clear, since latency is the same for all network types.

![Figure 7. Comparison of Push and Pull Strategies using class splitting.](image)

**State Transmission**

Finally, we discuss some limitations of the standard Java state transmission scheme that is used in all mobile agent toolkits today and which decreases the migration efficiency for data. As part of the migration process (outlined in Sec. 2), the agent’s state is serialized, i.e. all elements of the agent’s object closure are transferred into a flat byte stream. This can be seen as a push strategy, because all elements of the object closure are transmitted at once. With pure Java techniques it is not possible to realize a state transmission technique comparable to a pull strategy.

The advantages of a pull strategy for (parts of) the state are obvious and we have implemented the following approach in Kalong. An agent can decide to leave specific elements of its state (we call them data items) at its home agency, because of a low read (access) probability. When the data item is actually needed, it can be loaded on demand—comparable to a needed class. In addition, agents
can send back data items to their home agency, if it is known that they will not be used in future. The general effect of uploading data items to the home agency is very powerful and promising, but our experiments are still in the beginning. A fully automated approach to decide on access probability of data items seems to be very difficult. Therefore, at the moment we have only experience with manually decisions.

Conclusion and Outlook

The interest in mobile agents as a new design paradigm for distributed systems seems to have dwindled over the last years. The number of research groups working on mobile agent related research topics is becoming smaller. It is argued that mobile agents were not able to satisfy main expectations, for example regarding network traffic overhead. Vigna [24] states that mobile agents are very expensive and provide worse performance in the general case than other design paradigms, as for example remote procedure call or remote evaluation.

We agree with Vigna on the fact of poor performance, but we draw different conclusions. Instead of abandoning the concept of mobile agents, we propose to improve the migration process of mobile agents. In this paper, we have introduced the notion of migration efficiency to describe reasons for mobile agents’ poor performance. We have presented several drawbacks of Java-based mobile agents, which all result in a low migration efficiency. We have shown our solutions to all these problems and presented results of first experiments to show that our solutions significantly improve migration performance. We have implemented all our solutions in a new mobility model, named Kalong, which is available as software component that is ready to be integrated in most Java-based mobile agent toolkits. Kalong comes with some further features, which are outside the scope of this paper. For example, agents (or programmers) can dynamically set up code servers and Kalong has already implemented a few security protocols.

Some of our approaches work semi-automatically at the moment and make user intervention necessary to decide on method distribution, for example. Our future work will go into the direction of fully-automated approaches, where mobile agents learn over time and in co-operation with other agents of the same type to decide how to split and how to migrate in the most efficient way.

References


[22]  

[23]  

[24]  