

# Increasing the Migration Efficiency of Java-based Mobile Agents

Peter Braun, Ingo Müller, Ryszard Kowalczyk  
Swinburne University of Technology  
Faculty of Information and Communication Tech.  
Hawthorn, Victoria 3122, Australia

Steffen Kern  
Friedrich Schiller University Jena  
Department of Computer Science  
07743 Jena, Germany

## Abstract

*Mobile agents were introduced as a 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 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.*

## 1. Introduction

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*, i.e. their potential to save network load by moving code close to the data rather than moving data to the code as it is done in the client-server paradigm. This advantage of mobile agents has been scrutinized in the last years by many different research groups for different application domains. [6, 8, 9]. However, these papers also pointed to some situations, in which mobile agents have revealed severe disadvantages. As a consequence, a software designer has to carefully decide between both paradigms on a case-by-case basis [3]. We propose to supplement such a static design decision between agent migration and remote communication [7, 10] with techniques to reduce the migration overhead of mobile agents by increasing the *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 are used (read or writ-

ten) 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. A migration efficiency greater than 1 can make sense, since agents' code can be deployed or be cached in advance. It should be clear that the performance of a mobile agent is directly influenced by the migration efficiency.

## 2. Migration Strategies

In the following, we use the term *migration strategy* to describe how code and data are relocated in a mobile agent system. Whereas data relocation is always based on the Java serialization mechanism, code relocation is implemented differently in current mobile agent toolkits.

We can roughly distinguish between two code relocation 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. 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 in Sec. 4 that our intuition of *necessary* classes does not match the Java language definition.

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, are transmitted at once. 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.

It is not surprising that neither pure push nor pure pull strategies lead to the minimal network load and application performance in all cases [3]. Our thesis is that mobile agents should be able to adapt their migration strategy according to specific environmental parameters (code size of each class, the probability that a class is used at the next destinations, network bandwidth and latency, etc.) in order to increase the migration efficiency. In none of the available toolkits the agent (or agent programmer) can influence the migration strategy during runtime in order to select a migration strategy.

### 3. Adaptive Transmission of Code and Data

We have implemented a new mobility model, Kalong [3], which is available as a software component independent from any specific toolkit. We have already successfully integrated Kalong into the Tracy [2,3] and the Jade [4] toolkit. The Kalong mobility model can be seen as a virtual machine for agent migration, for it provides a basic set of commands to conduct the complete migration process. With Kalong, the migration strategy is no longer fixed but can be *programmed* by the agent during run-time. Kalong differs from current mobility models in three main aspects:

1. Kalong defines a new agent representation and new transmission units. In our model, mobile agents not only consist of an object state and their code, but have also an *external state*, which comprises of data items that are not part of the object state. Whereas the object state is still completely transmitted during an agent migration, external data items can be sent back to the agent's home agency and later be loaded from there again. A mobile agent's code is no longer transmitted in form of classes or JAR files, but we introduce a new transmission format that we call *code unit*.
2. Kalong defines two new agency types, additionally to the already known *home* and *remote* agencies as in current mobility models. We introduce a *code server* agency, from which an agent can download code on demand, and we introduce a *mirror* agency, which is an exact copy of an agent's home agency.
3. Kalong defines a new class cache mechanism, that not only prevents class downloading in the case of

pull strategies, but also code transmission in the case of push strategies.

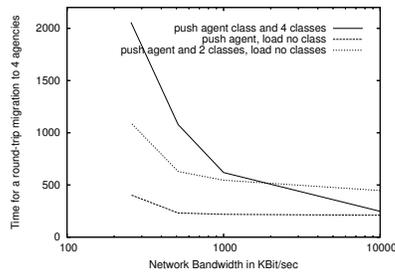
Basic commands of the Kalong virtual machine are, for example, sending an agent's state or sending an agent's code unit. Classes that were not sent during migration must be loaded (pulled) on demand later. This may result in any kind of mixture of push and pull strategies, for example, to only push the agent's main class and to dynamically load other classes. Additionally, the migration strategy also defines, which data items of the external state should be migrated to the next destination or, for example, sent back to the agent's home agency.

It should be obvious that by using these primitives for state and code transmission, it is possible to describe all migration strategies that we have introduced in the previous section. 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 strategy, we define that each class forms a single code unit and none of them is transmitted along agent's state.

The problem of deciding which classes should be pushed or pulled is subject of ongoing research and we experiment with various techniques to enable agents this decision. In [5] we have described a static profiling approach to determine the usage probability based on a byte code analysis. This analysis is comparable to the one we use for *class splitting*, see Sec. 6 and [5].

The second approach we have implemented is based on dynamic profiling. We assume an agent (or, in general, different agent instances of the same agent type) to execute the same task frequently, so that we can infer from class usage information gathered during one sample execution to succeeding executions. Dynamic profiling is implemented by annotating Java byte code with a counter for each class (a counter is a class variable), which is increased for every method invocation. Because class variables are not part of the serialized object closure, the agent must save these values before a migration to the object state. After one agent execution, the agent knows how often which of its classes has been used and it can decide on classes to push (those classes that have been used more often than a specific threshold) and those classes to be pulled during a subsequent execution.

In Fig. 1 we show the result of an experiment, in which an agent decides between a push and pull strategy during run-time. The experiment was done using the Tracy mobile agent toolkit in a Fast-Ethernet local area network with a theoretical bandwidth of 100 Mbit/sec. To measure the effect of different network qualities, we reduced the bandwidth manually to 256



**Figure 1. Comparison of different migration strategies in various network environments. Times are given in millisecond.**

Kbit/sec, 512 Kbit/sec, and 1 Mbit/sec for some experiments. We measured round-trip times of a mobile agent migrating to 4 agencies, each located on a different PC with a 1.4 GHz Athlon processor and 512 MB memory running Linux Fedora 3. The agent consists of a main class (code size 3566 Byte) and 4 auxiliary classes (code size 3822 Byte each). Each experiment was repeated 500 times and we only report mean values here. The top 5% of the measured times have been dropped to disregard those measurements messed up by the Java garbage collector. We set up an artificial runtime environment that consisted of three scenarios. In each scenario, the agent first migrates to collect dynamic profiling information using a pull strategy. For the second round, the agent then decides between pushing and pulling classes and we only report these times here. For the following, compare Fig. 1. The solid line shows the round-trip time in the case that the agent uses all classes, which are then pushed. The dotted line shows the case that only the agent's main class and two auxiliary classes have been used and therefore pushed (and no other classes pulled later). Finally, the dashed line shows the case that only the agent's main class is pushed (and no other classes pulled later).

From the experiment we can learn the following: At first, it is obvious to see that the difference between different migration strategies becomes smaller with increasing network bandwidth. Second, the differences between migration strategies are significant, although classes are small in our experiment.

#### 4. 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, we have found that the Java virtual machine loads all classes of object attributes 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, 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.

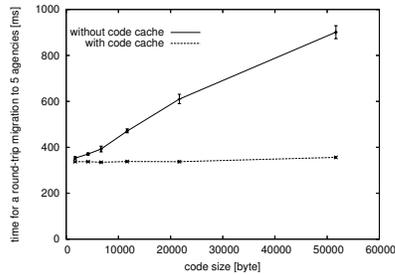
We have found a straightforward solution to this problem by changing the type of all object attributes to *Serializable* and performing necessary type casts when it is used. The effect is that now classes are only downloaded, if an object of this type is used. We have implemented this code rewriting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

#### 5. Code Caching

Code caching can be a powerful technique to increase migration efficiency. However, the existing Java mechanism has two shortcomings. At first, the Java class cache only prevents code downloading (pulling)—but not code pushing. Second, 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 [1], all agents are loaded by the same class loader. No agent class is loaded twice in Jade, which has the consequence that Jade cannot distinguish between different versions of the same agent 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.

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. We use MD5 *digests* to compare two classes unambiguously. Classes are not cached as part of a class loader object, but in a separate sub-component of Kalong.

Fig. 2 shows the result of an experiment, where an agent of one class and various code sizes processed an itinerary to 5 agencies twice (identical hardware parameters as described in Sec. 3, 100 Mbit/sec network). On its first tour, all code must be transmitted to each agency, whereas on the second tour, no code was trans-



**Figure 2. Difference in migration times with and without using class cache.**

mitted. The diagram shows the significant effect of code caching on processing times.

## 6. 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 to further increase migration efficiency, as 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. We cannot describe our approach here due to a lack of space and we refer to [5] for a detailed description of our implementation.

## 7. Conclusion and Outlook

In this paper, we have introduced the notion of *migration efficiency*. 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.

## References

- [1] F. Bellifimino, G. Caire, A. Poggi, and G. Rimassa. Jade – A White Paper. *EXP in search of innovation*, 3(3):6–19, 2003.
- [2] P. Braun, I. Müller, S. Geisenhainer, V. Schau, and W. R. Rossak. Agent migration as an optional service in an extendable agent toolkit architecture. In A. Karmouch, L. Korba, and E. Madeira, editors, *Proceedings of the*

*First International Workshop on Mobility Aware Technologies and Applications (MATA 2004)*, Florianopolis (Brazil), October 2004, volume 3284 of *Lecture Notes in Computer Science*, pages 127–136. Springer Verlag, 2004.

- [3] P. Braun and W. R. Rossak. *Mobile Agents—Basic Concept, Mobility Models, and the Tracy Toolkit*. Morgan Kaufmann Publishers, 2005.
- [4] P. Braun, D. Trinh, and R. Kowalczyk. Integrating a new mobility service into the Jade agent toolkit. In A. Karmouch and S. Pierre, editors, *Proceedings of the Second International Workshop on Mobility Aware Technologies and Applications (MATA 2005)*, Montreal (Canada), October 2005, *Lecture Notes in Computer Science*. Springer Verlag, 2005.
- [5] S. Kern, P. Braun, C. Fensch, and W. R. Rossak. Class splitting as a method to reduce the migration overhead of mobile agents. In R. Meersman, Z. Tari, and A. Corsaro, editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004, Agia Napa (Cyprus), October 2004, Proceedings, Part II*, volume 3291 of *Lecture Notes in Computer Science*, pages 1358–1374. Springer Verlag, 2004.
- [6] A. Outtagarts, M. Kadoch, and S. Soulihi. Client-Server and Mobile Agent: Performances Comparative Study in the Management of MIBs. In A. Karmouch and R. Impy, editors, *Mobile Agents for Telecommunication Applications, Proceedings of the First International Workshop (MATA 1999)*, Ottawa (Canada), October 1999, pages 69–81. World Scientific Pub., 1999.
- [7] G. P. Picco. *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility*. PhD thesis, Politecnico di Torino (Italy), 1998.
- [8] A. Puliafito, S. Riccobene, and M. Scarpa. Which paradigm should I use? An analytical comparison of the client-server, remote evaluation and mobile agent paradigms. *Concurrency and Computation: Practice and Experience*, 13(1):71–94, 2001.
- [9] G. Samaras, M. D. Dikaiakos, C. Spyrou, and A. Livros. Mobile Agent Platforms for Web-Databases: A Qualitative and Quantitative Assessment. In D. S. Milojicic, editor, *Proceedings of the First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs (USA), October 1999, pages 50–64. IEEE Computer Society Press, 1999.
- [10] M. Straßer and M. Schwehm. A performance model for mobile agent systems. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Las Vegas (USA), volume 2, pages 1132–1140. CSREA Press, 1997.