Enhancing Agent Mobility Through Resource Access Policies and Mobility Policies

Alejandro Zunino[1] , **Cristian Mateos**[1] , **Marcelo Campo**[1]

[1] CONICET - ISISTAN Research Institute
Fac. de Ciencias Exactas, Dpto. Computación y Sistemas
Universidad Nacional del Centro
Campus Univ. Paraje Arroyo Seco - Tandil - (B7001BBO)
Buenos Aires, Argentina

azunino@exa.unicen.edu.ar

Abstract. *MoviLog is a novel programming language that aims at reducing the effort of programming mobile agents. MoviLog is based on the concept of Reactive Mobility by Failure (RMF), a mechanism that migrates a mobile agent when it tries to access a resource that is not located at the current site. This paper presents an extension of RMF that allows the programmer to adapt the mechanisms used by RMF for deciding when and where to move an agent. Experimental results showing gains in network usage and performance with respect to client/server and traditional mobility mechanisms are also reported.*

1. Introduction

Mobile agents, autonomous software capable of migrating their execution between network hosts in order to achieve their users' goals, have been proposed as an alternative to solve some of the problems of large distributed systems (Gray et al., 2001). Mobile agents are different from non-mobile or stationary software in one very important aspect: mobile agents are *location aware*, this is, mobile agents know the network topology and its resources, and in order to take advantage of them, mobile agents move their execution from host to host. As a consequence, mobile agents are able to choose, for example, whether to remotely access to a non-local resource or migrate to the place where the resource is located, depending on the cost of remotely interacting with the resource and the cost of migrating to its location.

In recent years, mobile agents have shown advantages with respect to traditional paradigms such as client/server. This motivated the development of many tools for programming mobile agents (Nelson, 1999; Silva et al., 2001; Fuggetta et al., 1998; Thorn, 1997). Despite the advantages of mobile agents and the wide availability of tools, their usage is still limited to small applications and academic prototypes (Kotz et al., 2002; Fradet et al., 2000; Picco et al., 1997). In part, this is caused by the complexity of mobile agent development with respect to non-mobile systems. Indeed, developers have to provide mobile agents with mechanisms to decide when and where to migrate. Therefore, though agents location awareness may be very beneficial if properly used, it also adds further complexity to their development (Picco et al., 1997; Silva et al., 2001).

MoviLog (Zunino et al., 2003; Amandi et al., 2005) is a programming language whose goal is to reduce the effort of developing intelligent mobile agents. MoviLog is based on the concept of *Reactive Mobility by Failure* (RMF). The idea behind RMF is to assist mobile agents on making decisions on *when* and *where* to migrate. In order to retain agents' autonomy, RMF only interferes with their normal execution when it detects an *m-failure*. An *m*-failure occurs when a mobile agent accesses to a specially declared resource that is unavailable at the current site. In addition, RMF provides the same advantages as traditional mobility, namely reduced network usage, better scalability, robustness, etc.

To sum up, MoviLog allows programmers to delegate decisions about when and where to migrate an agent, but doing so implies that the programmers lose control of how these decisions are made. In some cases this may lead to performance problems. Consider, for example, a situation where a mobile agent located at a host *o* needs a database that is replicated at two hosts *a* and *b*. At this point, RMF moves the agent to one of these hosts, for example *a*. Now, let us suppose that the agent needs a small file located at *o*, thus RMF migrates the agent from *o* to *a*. Note that depending on the size and time needed for migrating the agent it could be more convenient to transfer the file from *o* to *a* instead of migrating the agent.

Up to now MoviLog had problems with this type of situation because RMF was not able to take into account application specific information that might lead to better decisions. This paper describes an extension of MoviLog aimed at enhancing the performance of mobility by allowing the programmer to tailor RMF. The idea of the extension is that programmers should be able to specify *policies*, this is, the mechanisms that the algorithms for automatizing mobility use for deciding. These policies help the automatic parts of MoviLog in making decisions taking into account aspects such as agent size, network load, CPU speed, etc., when deciding whether to migrate or not and where to migrate.

The paper is organized as follows. The next section introduces the concept of RMF. Section 3 describes the extensions of RMF. Then, experimental results are reported in Section 4. In Section 5 the most relevant related work are analyzed. Finally, concluding remarks are presented in Section 6.

2. MoviLog and Reactive Mobility by Failure

Mobility can be either *proactive* or *reactive* according to whom triggers it:

- *proactive*: the time and destination for migrating are autonomously decided by the migrating agent.
- *reactive*: migration is triggered by an entity external to the migrating agent.

Most existing mobile agent platforms support the former type of mobility only. On the other hand, RMF (Zunino et al., 2005) combines both types to automatize decisions on when and where to migrate mobile agents, thus reducing their development effort.

RMF is based on the assumption that mobility is orthogonal to the rest of capacities that an agent may possess (Nwana, 1996): reasoning, reactivity, learning, mobility, etc. RMF exploits the conceptual independence among agent capacities at the level of implementation by separating agent functionality into two classes: *stationary* and *mobile* functionality. Stationary functionality is concerned with those actions executed by agents at each site of a network. Mobile functionality is mainly concerned with making decisions about when and where to move. RMF takes advantage of this separation by allowing the programmer to focus his efforts on the stationary functionality.

Before going into further details we will first define several important concepts. In the rest of the paper we will refer to mobile agents in MoviLog as *Brainlets*. The runtime platform residing at each host that provides support for executing Brainlets will be called a *MARlet*. A set of MARlets such as all of them know each other will be called a *logical network*. A logical network groups MARlets belonging to the same application or a number of closely related applications. In addition, MARlets are able to provide resources such as databases, procedures or Web Services to Brainlets.

A Brainlet consists of a sequence of Prolog clauses and a possibly empty sequence of *protocols*. As stated above, RMF is a mechanism that acts when an *m*-failure occurs. An *m*-failure can be defined as the failure of a Prolog goal declared as a protocol. In other words, protocols describe all those Prolog predicates whose failure may trigger mobility. The failure of any other goal will not trigger mobility.

When a Brainlet produces an *m*-failure, the MARlet reactively migrates the Brainlet to other MARlet with clauses with the same protocol as the one that failed. If more than one MARlet possess clauses with that protocol, then the local MARlet builds and incrementally updates an itinerary to visit those MARlets. Once the Brainlet has migrated, its execution is resumed at the destination. As a consequence mobility is transparent to the Brainlet.

It is worth noting that the Brainlet does not decide neither the time to migrate nor its destination. The migration is triggered by an *m*-failure, then the destination is dynamically selected by the MARlet by communicating with its peers in the logical network. As a result, even when the Brainlet knows nothing about mobility, RMF is able to determine when and where to migrate the Brainlet. Indeed, the only data that a Brainlet is required to specify about mobility are the predicates whose failure is to be treated as an *m*-failure.

The next subsections describe further details of RMF.

2.1. Brainlets

Brainlets are Prolog-based mobile agents that support proactive and reactive mobility. In both cases, mobility is supported by a *strong* migration mechanism. By *strong* we mean the ability of a mobile agent run-time system to allow migration of both the code and the execution state of a mobile agent. In opposition, *weak* migration is not capable of transferring the execution state of a mobile agent and hence it *forgets* the point at where it was executing before migrating. Despite the obvious shortcomings of the second type of migration, it is widely supported by most mobile agent platforms because it can be implemented in a simpler manner than strong migration. On the other hand, though strong mobility is difficult to implement, their usage to program mobile agents is much simpler than weak migration (Silva et al., 2001).

To sum up, Brainlets are mobile agents composed of the following parts:

- *code*: comprises a sequence of Prolog clauses implementing the agent behavior.
- *data*: consists of a sequence of Prolog clauses.
- *execution state*: is composed of one or more Prolog threads. Each thread comprises a program counter, variable bindings and choice points (for handling backtracking). Execution state is preserved on migration.
- *protocols*: describe all those Prolog predicates whose failure may trigger mobility.

2.2. Protocols

A protocol is a declaration with the syntax `protocol(functor, arity)` that instructs the RMF run-time to treat the failure of goals with the form `functor(arg[1], ... , arg[arity])` as *m*-failures. Protocols are used for two reasons:

- from a Brainlet point of view: to let the programmer control the points of a Brainlet code that may trigger reactive mobility.
- from a MARlet point of view: to describe clauses, or more generically the interface for accessing resources, available in a logical network. RMF works when

Table 1: Three MARlets and their clauses

M_1	M_2	M_3
hd(#123, eide, wd, 5400, 40, 72)	hd(#78, scsi, ibm, 15000, 36.7, 187)	hd(#22, scsi, seagate, 7200, 36, 210)
hd(#23, eide, maxtor, 7200, 40, 79)	hd(#33, eide, quantum, 7200, 20, 54)	hd(#44, eide, panasonic, 7200, 30, 582)
hd(#78, scsi, ibm, 15000, 36.7, 187)	hd(#45, eide, ibm, 5200, 30, 114)	

an *m*-failure occurs by searching in the logical network for all the MARlets providing clauses with the same protocol as the goal that *m*-failed. Protocols enable MARlets to describe those clauses or resources that they provide.

A simple example will clarify the concepts introduced up to now. We will first show a Prolog program. Then, we will define a protocol to show how the program becomes a mobile agent. Let us consider the following Prolog program:

```
preferred(Id) :- hd(Id,eide,_,_,_,_).
 preferred(Id) :- ...
?-findall( Id+Price,
     ( hd( Id, Type, Brand, RPM, MB, Price ), preferred(Id) ),
     L ).
```

`findall` searches the Prolog database for clauses `hd` with six arguments, representing hard disks, satisfying a number of preferences given by a user. The result of `findall` is a list `L` containing pairs (*Id*, *Price*), where *Id* is the serial number of the hard disk and *Price* is its price. The predicate `preferred(Id)` evaluates to `true` if the hard disk identified by *Id* matches a number of preferences such as brand, capacity, speed and price.

In order to explain the execution of the program we will consider a Prolog database containing three clauses (column M_1 of the table 1). If we execute the program with those clauses we obtain a list [#123+72, #23+79], stating that the hard disks #123 and #23 whose prices are 72 and 79, respectively, match users' preferences.

Now we will modify the program to use RMF for searching in the three MARlets for hard disks. The idea is to trigger mobility upon *m*-failures of predicates `hd` and hence forcing the program to visit M_1, M_2 and M_3. The modified code is:

```
PROTOCOLS
  protocol(hd, 6).
CLAUSES
  preferred(Id) :- hd(Id,eide,_,_,_,_).
  preferred(Id) :- ...
  ?-findall( Id+Price,
     ( hd( Id, Type, Brand, RPM, MB, Price ), preferred(Id) ),
    L )
```

The code is divided into two sections: *protocols* and *clauses*. The first section contains protocol declarations. The second section contains the code and data of the Brainlet.

As in the previous example, `findall` searches for hard disks satisfying a number of preferences. The code behaves the same as the first example up to the point when `findall` tries to evaluate `hd` for the fourth time. In this case, the evaluation of `hd` will fail, but considering that `hd` has been declared as a protocol, an *m*-failure will occur and hence RMF will find a MARlet providing clauses `hd` to migrate the Brainlet and to try to reevaluate the goal there. As shown in table 1, there are two options, either M_2 or M_3. Let us assume that RMF selects M_2. Then, after the migration of the Brainlet to M_2, `findall` continues searching hard disks until no more options are available. In this point an *m*-failure will occur and RMF will select M_3. After finding hard disks at M_3, `hd` will fail again. In this case there will be no more options left for migrating the Brainlet. Then, it will be returned[1] to its origin (M_1) by the MARlet M_3. Finally, the result of `findall` will be [#123+72, #23+79, #33+54, #45+114, #44+582].

As shown in the example, protocols enable the programmer to delegate mobility decisions on RMF. At the same time, the programmer can combine proactive mobility with RMF as needed by using the predicate `moveTo(S)` that migrates the Brainlet to a MARlet whose name is `S` (see (Zunino et al., 2003) for further details on proactive mobility on MoviLog).

Up to now we have described how RMF handles mobility automatically. The next section is concerned with the extensions for tailoring RMF.

3. Resource Access Policies and Mobility Policies

Despite the positive results RMF has shown (Zunino et al., 2003), some problem may arise due to the lack of information RMF has about the application being executed.

For example, consider a Brainlet that causes an *m*-failure thus RMF migrates it to a remote MARlet. Once there, the Brainlet just solves the failure by using a clause available at the MARlet database and returns to its origin. The problem here is that the two migrations of the Brainlet are certainly more expensive in bandwidth and time than the cost of copying the clause needed to solve the failure from the remote MARlet to the origin.

A similar situation occurs when, after an *m*-failure, a clause is available at several MARlets. The first problem RMF has to solve is where to migrate the executing Brainlet. It may decide blindly thus the Brainlet may end up running on a heavily loaded MARlet, or travel through several very slow network links, or worst, it may choose a site that charges for CPU usage.

Despite RMF is a step in making mobility easy to use, delegating mobility decisions on RMF may be too expensive in terms of performance in some cases. In order to solve these problems we extended RMF to support *policies*, these are, mechanisms provided by the programmer to adapt RMF to his requirements. Policies can be classified into two types:

- *resource access policies*: in the previous example, *m*-failures always cause the migration of the executing Brainlet. However, *m*-failures can be treated either by copying clauses from other MARlets to the local one, or by remotely accessing clauses stored at other MARlets of the network, depending on several factors such as network traffic, resource usage, etc.
- *mobility policies*: when more than one MARlet offer the protocol of the goal that *m*-failed it may be necessary to visit some or all of them for reevaluating the goal. In addition, the order for visiting these MARlets may be important. For example, it may be convenient to visit the sites according to their speed, CPU load or availability.

The next sections describe each type of policy in detail.

[1]After a successful evaluation of a predicate that *m*-failed a Brainlet does not return to its origin. It returns if it finishes its execution, fails (no more alternatives are available) or RMF decides so.

3.1. Resource Access Policies

The extended RMF is able to both migrate a Brainlet to a remote MARlet or transfer clauses from a remote MARlet to where the Brainlet is hosted. This decision is made by considering two factors:

- *The type of resources*: protocols describe an interface for accessing a resource. The type of a resource determines whether the resource is *transferable* or *non-transferable*, i.e. whether it can be migrated over the network or not. For example, let us consider a protocol that describes a number of clauses that require a local printer for executing. Obviously, these clauses cannot be easily transferred because of their dependence on the non-transferable local printer.
  Protocol declarations may include a third argument indicating that the resources are transferable. Then, the programmer could associate a *migration policy* to these resources.
- *A resource access policy*: is a user-defined mechanism used by RMF to determine whether to migrate a Brainlet or fetch resources from remote MARlets. For example, a simple policy is to migrate a Brainlet if the network traffic generated by migrating the Brainlet is less than the approximated traffic for fetching clauses.

When an *m*-failure occurs on a protocol marked as transferable, RMF obtains and evaluate the resource access policy associated with the protocol. From this evaluation RMF decide whether to fetch clauses from remote MARlets or migrate the Brainlet.

In the example described in Section 2.2 we could use a resource access policy to determine whether to migrate the agent or fetch all the clauses `hd` with six arguments from a remote MARlet:

```
PROTOCOLS
  protocol(hd, 6, transferable).
  accessPolicy(hd, 6, (estTrM(A, D, T1),
    estTrFAll(hd, 6, D, T2), T1 < T2 ) ).
CLAUSES
   preferred(Id) :- hd(Id,eide,_,_,_,_).
   preferred(Id) :- ...
  ?-findall( Id+Price,
       ( hd( Id, Type, Brand, RPM, MB, Price ), preferred(Id) ),
     L )
```

`accessPolicy` states that in case of an *m*-failure the Brainlet migrates if the network traffic $T1$ required for migrating the agent A to MARlet D, denoted `estTrM(A, D, T1)`, is less than the estimated network traffic $T2$ required for fetching all the clauses `hd` from MARlet D, denoted `estTrFAll(hd, 6, D, T2)`. Otherwise, the clauses are fetched.

3.2. Mobility Policies

The goal of a mobility policy is to build an itinerary for a Brainlet when the protocol of the goal that *m*-failed is available at more that one MARlet of the logical network. The idea is to try to probe the goal by using a depth first search strategy similar to the one used by Prolog, but taking into account the location of the code. In other words, to probe the goal ensuring completeness it may be necessary to visit more that one MARlet.

A mobility policy provides a mean for the programmer to specify decision mechanisms for ordering a set of MARlets for building an itinerary. The basis of the mechanism consists on associating a *mobility policy* with each protocol. The policy is used for determining the next destination for reevaluating a goal given a set of MARlets offering the same protocol as the goal. For example, in a CPU bound application might be useful to visit MARlets according to the CPU load at each site, other types of applications may use a policy based on the network load.

For the previous example we could specify a policy for visiting MARlets according to the speed of the network link between the current MARlet and the destination as follows:

```
PROTOCOLS
  protocol(hd, 6, transferable, shortestMoveTimePolicy[2]).
  ...
```

Examples of other policies are cpuLoadPolicy, freeMemoryPolicy, randomOrderPolicy. In addition, the programmer can define new policies in Prolog or Java. It is worth noting that both protocols and policies can be manipulated at run-time by the programmer.

4. Experimental Results

In order to validate the approach we have implemented a distributed algebraic expression solver using RMF, proactive mobility and a messages without mobility (traditional client/server). The application consists of two types of Brainlets:

- *server*: a server is a Brainlet that provides services for solving simple arithmetic operations such as +, -, /, or * thus they are not able to solve compound operations. Each server is able to solve a single type of arithmetic operations. In addition, each server is statically assigned to a specific host of the network, thus servers are not allowed to migrate during their execution.
- *client*: a client is a Brainlet that knows nothing about solving simple arithmetic operations, but it is able to split compound expressions into simpler operations by applying associativity rules. For example:

  $$\frac{2*2}{3-1}+10-7$$

  can be rewritten as a number of binary operations: (((2*2)/(3-1))+10)-7. Then, these binary operations can be solved by servers.
  Since servers may reside in different sites than clients, and remote communications are not allowed (except in the no mobile solution), clients may migrate in order to request the resolution of a binary operation. When two ore more servers located at different sites are able to solve the same required operation, clients try to balance the CPU load of these sites by migrating to the less loaded site.

Seven different tests were run on a 100 Mbps LAN with 4 Pentium II PCs using Java 1.4.2 and Windows 2000. For RMF we used a policy that migrates agents to sites with low CPU load. Figure 1 shows the average running time and network traffic for 10 runs of each test.

Proactive mobility performed poorly due to the unnecessary large number of migrations and suboptimal itineraries. On the other hand, the performance of the stationary solution was in the middle between RMF and proactive mobility. RMF performed remarkably well because it was able to greatly reduce the number of migrations and traffic with respect to the other two implementations. It is worth noting that the stationary solution has disadvantages when clients and servers interact often and are hosted at different sites.

[2]This policy uses IP ICMP echo requests to determine the network delay between two hosts.

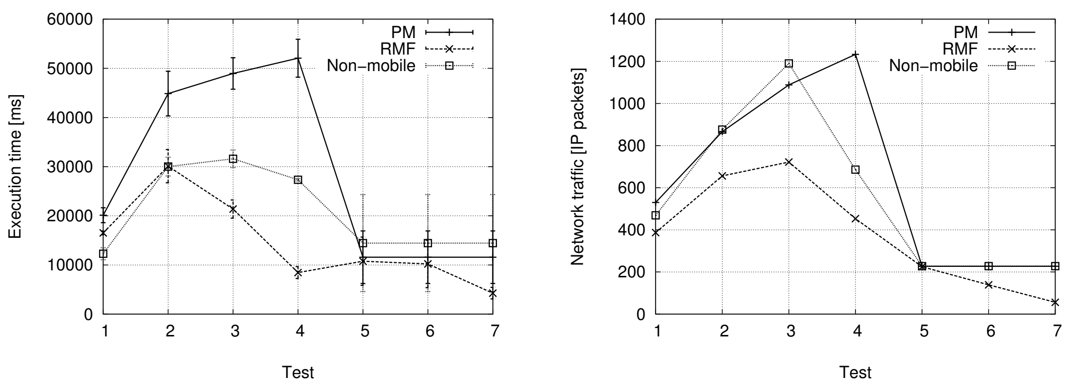

Figure 1: Experimental results

In this cases mobility uses less network traffic because remote interactions are avoided by migrating clients.

It is worth noting that the size of the agent implemented with RMF was 23% smaller than the one implemented with proactive mobility. This shows the clear advantage in terms of reduced programming effort that RMF has. Furthermore, less code implies that migration is faster and less network bandwidth is used for migrating agents.

The importance of these results are twofold. On the one hand, they show that by using RMF, agents are smaller due to the simplification of the code for handling mobility. On the other hand, the results suggest that agents using RMF are faster and use network resources more efficiently than traditional proactive mobile agents. All in all, by extending RMF with policies we were able to obtain better results than proactive mobility and no mobility.

5. Related Work

There are many tools for supporting mobile agent development (Tarau, 1998; Picco, 1998; Johansen et al., 1995; Acharya et al., 1997; Thorn, 1997). However, most of them only provide rudimentary mechanisms for handling mobility based on weak and proactive migration mechanisms. As a consequence, programmers are in charge of handling mobility, this is, providing code for determining when and where to migrate an agent.

Some initial attempts to provide high level constructs for handling mobility are Concordia (Wong et al., 1997), Aglets (Lange and Oshima, 1998) and Ajanta (Tripathi et al., 2002)

Concordia was one of the firsts platforms to use itineraries. The idea is to provide each agent with a list of sites to visit and a task to perform at each site. In this way, Concordia uses a very simple approach for managing mobility and reducing programming effort. On the other hand Concordia uses a weak migration mechanism thus the programmer has to adopt a rather difficult event driven approach for programming mobility (Silva et al., 2001). The same applies to all of the three tools described as well as most Java-based platforms for mobile agent development. In opposition, MoviLog uses a strong mobility mechanism that is transparent and very easy to use.

Aglets introduced the concept of mobility patterns: recurring solutions that appear multiple times in mobile systems. Aglets defines a number of predefined patterns and provides implementations for these patterns. Examples of patterns are:

- *meeting*: several agents are required to meet at a specific host of a network in order to exchange messages

- *messenger*: an agent carries a message from one agent to another.
- *slave*: a master agent delegates a task to a slave agent. The slave follows an itinerary, executing a task in each site.

Despite the benefits in terms of development effort these patterns provide, the programmer is still in charge of handling most mobility decisions. Besides, the weak migration mechanism used by Aglets has the same problems as Concordia.

Ajanta takes the concept of itineraries a step further by providing migration patterns. A migration pattern is an abstract migration path for an agent. Examples are loop, selection, split and join. Itineraries are composed of a number of migration patterns. At each step of an itinerary an agent may move, perform a task, create or destroy other agents. These constructs provide powerful tools for building mobile agent systems with complex itineraries. However, these itineraries are statically defined instead of automatically as RMF does. As a consequence, Ajanta has problems handling highly dynamic networks as the Internet.

The main advantage of our approach is its capability for automatically handling mobility while being more efficient than traditional proactive mobility. In addition, because programmers are not required to provide code to handle mobility, RMF requires less code. As a consequence, agents are smaller, require less network bandwidth to migrate and are easier to develop, understand and maintain.

6. Conclusions

In this paper we have described an extension of RMF aimed at enhancing mobility efficiency by allowing the developer to adapt the mechanisms used by RMF for deciding when and where to migrate agents.

The approach has been implemented and compared with client/server and proactive mobility by using a distributed algebraic expression solver. The results are very encouraging since show important gains in performance and reduced network bandwidth. More experiments are being conducted in order to confirm these results in other applications.

We are exploring an extension of RMF for handling remote invocations, in addition to mobility and fetching. In essence, the idea is that given an *m*-failure, RMF should be able to do whatever is necessary to solve that *m*-failure. In this context, there are three alternatives: 1) move the Brainlet where the resource is located, 2) move the resource where the Brainlet is located or 3) use some remote invocation mechanism for using the resource. Note that 2) is only possible if the resource is transferable while 3) is possible if the resource accepts remote calls, for example, a shared printer, a Web server or a database. From this research we have already obtained some interesting results in the domain of Web Services (Mateos et al., 2005).

References

Acharya, A., Ranganathan, M., and Saltz, J. (1997). Sumatra: A Language for Resource-aware Mobile Programs. In Vitek, J. and Tschudin, C., editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 111–130. Springer-Verlag: Heidelberg, Germany.

Amandi, A., Campo, M., and Zunino, A. (2005). Javalog: A framework-based integration of java and prolog for agent-oriented programming. *Computer Languages, Systems & Structures*, 31(1):17–33.

Fradet, P., Issarny, V., and Rouvrais, S. (2000). Analyzing non-functional properties of mobile agents. In *Proc. of Fundamental Approaches to Software Engineering, FASE'00*, Lecture Notes in Computer Science. Springer-Verlag.

Fuggetta, A., Picco, G. P., and Vigna, G. (1998). Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361.

Gray, R. S., Cybenko, G., Kotz, D., and Rus, D. (2001). Mobile agents: Motivations and state of the art. In Bradshaw, J., editor, *Handbook of Agent Technology*. AAAI/MIT Press.

Johansen, D., van Renesse, R., and Schneider, F. B. (1995). An introduction to the TACOMA distributed system. Technical Report 95-23, Department of Computer Science, University of Tromsø, Tromsø, Norway.

Kotz, D., Gray, R., and Rus, D. (2002). Future directions for mobile agent research. *IEEE Distributed Systems Online*, 3(8).

Lange, D. B. and Oshima, M. (1998). *Programming and Deploying Mobile Agents with Java Aglets*. Addison-Wesley, Reading, MA, USA.

Mateos, C., Zunino, A., and Campo, M. (2005). Integrating intelligent mobile agents with web services. *International Journal of Web Services Research*, 2(2).

Nelson, J. (1999). *Programmong Mobile Objects With Java*. Wiley.

Nwana, H. (1996). Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244.

Picco, G. (1998). μCode: A Lightweight and Flexible Mobile Code Toolkit. In Rothermel, K. and Hohl, F., editors, *Proceedings of the 2nd International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 160–171. Springer-Verlag: Heidelberg, Germany.

Picco, G. P., Carzaniga, A., and Vigna, G. (1997). Designing distributed applications with mobile code paradigms. In Taylor, R., editor, *Proceedings of the 19th International Conference on Software Engineering*, pages 22–32. ACM Press.

Silva, A., Romao, A., Deugo, D., and Mira da Silva, M. (2001). Towards a Reference Model for Surveying Mobile Agent Systems. *Autonomous Agents and Multi-Agent Systems*, 4(3):187–231.

Tarau, P. (1998). Jinni: a lightweight java-based logic engine for internet programming. In Sagonas, K., editor, *Proceedings of JICSLP'98 Implementation of LP languages Workshop*, Manchester, U.K. invited talk.

Thorn, T. (1997). Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239.

Tripathi, A. R., Karnik, N. M., Ahmed, T., Singh, R. D., Prakash, A., Kakani, V., Vora, M. K., and Pathak, M. (2002). Design of the Ajanta System for Mobile Agent Programming. *Journal of Systems and Software*. to appear.

Wong, D., Paciorek, N., Walsh, T., DiCelie, J., Young, M., and Peet, B. (1997). Concordia: An infrastructure for collaborating mobile agents. In *First International Workshop on Mobile Agents (MA'97)*, pages 86–97.

Zunino, A., Campo, M., and Mateos, C. (2003). MoviLog: A platform for prolog-based strong mobile agents on the WWW. *Inteligencia Artificial, Revista Iberoamericana de I.A.*, 4(21):83–92. ISSN 1337-3601.

Zunino, A., Mateos, C., and Campo, M. (2005). Reactive mobility by failure: When fail means move. *Information Systems Frontiers. Special Issue on Mobile Computing and Communications*. to appear.