In: Computer Middleware: Architecture, Advantages and Applications ISBN
Editor: Unknown © 2010 Nova Science Publishers, Inc.

Chapter X

A Semi-automatic, Malleable Mobility Model for Rapid Prototyping of Mobile Agent Applications

Alejandro Zunino, Cristian Mateos and Marcelo Campo
ISISTAN Research Institute,
Universidad Nacional del Centro de la Provincia de Buenos Aires,
Tandil (B7001BBO), Buenos Aires, Argentina.
Also Consejo Nacional de Investigaciones Científicas y Técnicas
(CONICET).

Abstract

Mobile agents have been successfully used for building massively distributed systems. In spite of the advantages the paradigm has shown, mobile agents are still somewhat underrated and hard to develop. Consequently, we proposed the Reactive Mobility by Failure (RMF) mobility model [1] for simplifying mobile agent adoption and development. RMF enables the developer to non-intrusively delegate certain decisions about agent mobility to the underlying executing middleware. However, in its current shape, the model is not able to consider application-specific context, which may be helpful for making better decisions about mobility regarding execution performance and network usage. In this paper, we describe an extension of RMF aimed at improving the efficiency of RMF-based applications by allowing the programmer to tailor RMF according to his application requirements. In essence, the contribution of this paper is to show that it is possible to automate some mobility decisions via RMF, while keeping high levels of flexibility –the points of a mobile agent code at which the developer is allowed to use mobility– and performance through the addition of custom mobility decisions. We have developed a Prolog-based prototype of our extended RMF model to enable the rapid implementation of mobile applications. Experimental results showing the advantages of the approach with respect to related approaches are also reported.

1. Introduction

A mobile agent is a computer program able to migrate from site to site within a network to carry out one or more tasks on behalf of a user [2]. On each site, a mobile agent interacts with stationary service agents and other resources to accomplish its tasks. Conceptually, mobility enables agents to move to the specific site where a resource (e.g. a data source) is located, thus reducing remote interactions and therefore execution time and network latency, or to visit a suitable site to perform a CPU-intensive computation, thus improving throughput. Mobile agents have shown advantages in terms of flexibility, scalability and mainly reduced network bandwidth with respect to traditional non-mobile software [3]. For example, a user using a Personal Digital Assistant or a cell phone with an expensive and slow Internet connection could send a mobile agent to perform some processing on a connected server. The user can then disconnect from the Internet while the mobile agent is at the server, thus saving money, battery and time. After a while, the user could reconnect to receive the agent.

Mobile agents have been successfully employed in a diversity of areas such as network management [4, 5], distributed information retrieval [6, 7, 8], mobile computing [9, 10] and Grid computing [11, 12, 13], just to name a few. Unfortunately, despite these positive experiences, the benefits of mobile agent technology are often eclipsed by its inherent “difficult development” characteristic [14, 15, 16]. Indeed, current approaches force application programmers to manually bundle API directives for controlling mobility aspects such as when and where to move an agent directly into its code [17], which naturally mixes up with the code implementing the pure agent behavior. This has some evident disadvantages from a software engineering perspective in the sense that good values for important software quality attributes such as modifiability and testability are more hard to obtain. Though mobile agents have interesting features for building distributed systems, this fact makes mobility-based software development more difficult than its non-mobile alternative [17], which in turn has hindered the widespread adoption of mobile agents for distributed computing. In this line, addressing the problem of simplifying the development of mobile applications has gained much attention, as mobility seems to have finally found its long-awaited "killer application" in the development of applications in large-scale contemporary distributed environments, namely Grids and Clouds [18, 19].

Apart from the clear disadvantages of having the code in charge of performing mobility scattered in the code implementing an agent's behavior, most approaches for developing mobile applications rely on proactive mobility, a programmatic, explicit form of mobility that is not suitable for these new environments. Essentially, these massively distributed settings are highly dynamic in the sense that hosts usually enter and leave the network often, causing proactive mobility models, which are commonly based on static itineraries, not applicable. A typical problem in this respect is how to effectively deal with itinerary update when hosts may randomly leave or enter the network, and accept or deny mobile agents at will, while keeping programming complexity low.

Reactive Mobility by Failure (RMF) has been proposed as an approach to facilitate mobile agent development [1]. RMF aims at making mobility almost invisible by supporting it at the middleware level. RMF intervenes with the normal execution of a mobile agent in specific points of its code to detect *m-failures*. An *m*-failure is an attempt for accessing a resource (data, libraries, services, etc.) that is unavailable at the local executing site. RMF is responsible for moving the mobile agent causing the *m*-failure to a machine with the required resource and then resuming the agent execution, while dealing with resource as well as host volatility. In the end, mobility is more transparent to the application programmer and the agent code is simpler, shorter and cleaner [1].

RMF has been implemented in MoviLog [20], a programming language based on an integration of Prolog and Java. Experimental results suggest that MoviLog reduces the code necessary to implement mobile applications compared with proactive mobility [1], the mechanism which most mobile agent tools are based on. When employing proactive mobility, programmers embed migration directives into the agent code. In these experiences, mobile agent size and network traffic were reduced by using RMF. However, in some situations, MoviLog moves an agent more times than necessary causing excessive execution overheads and network traffic. The cause of this problem is that mobility is always selected as the target mechanism for handling failures, however resource fetching[1] may be better suited depending on the characteristics of the application at hand, such as the order in which a set of needed resources are accessed by the agent or the number of accesses. For instance, let us suppose that the agent needs to query a database located at some site S. Depending on the number of accesses to the database it could be more convenient to transfer the agent to S instead of querying the database remotely.

The problem is that by hiding most mobility details from the programmer, MoviLog is not able to consider application-specific context that may be helpful for making better decisions about mobility, this is, when to exploit it and when not. In the above example, this context is given by the number of accesses to the database, which is information that, unless explicitly indicated by the programmer, is not available to MoviLog. Precisely, this paper describes an extension of RMF aimed at improving its performance by allowing the programmer to adapt RMF. The idea of the extension is that programmers should be able to specify *policies* conveying application-specific contextual information for adapting the mechanisms that RMF uses for automating mobility. Roughly, policies specify rules for deciding when and where to move an agent based on both this information and current execution conditions.

While most approaches for handling mobility embed migration directives into behavioral code, policies are separated from agent code. As a result, the code implementing an agent's functionality does not get mixed with its mobility-related code. This practice, which can be viewed as a form of separation of concerns, makes mobile agent code easier to develop, maintain and understand [21, 17]. In opposition to RMF and thus to MoviLog, where the programmer has little flexibility to control how agent mobility is managed, the main contribution of this paper is to show that it is feasible to automate many mobility decisions at the middleware level while sacrificing as little flexibility

[1]Instead of moving an agent to the host where a required resource is hosted, the resource is moved or copied to the agent's location. Java Applets and ActiveX controls are two examples of technologies based on this paradigm.

and performance –in terms of execution speed and network traffic– as possible.

The rest of the paper is organized as follows. The next section introduces the concept of RMF. Section 3. describes the extensions made to RMF to support policies. Section 4. reports experimental results that were carried out to validate our approach. Section 5. discusses the most relevant related works. Finally, Section 6. presents concluding remarks.

2. MoviLog and Reactive Mobility by Failure

Mobile agents in MoviLog can use two classical forms of mobility [22, 18]: proactive (or subjective migration) and reactive (or forced migration). Proactive mobility means that migration is initiated from inside the agent's code by invoking a move sentence. Reactive mobility is triggered by an entity external to the agent. In both cases, *strong* migration supports mobility. 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 cannot transfer the execution state of a mobile agent. Therefore, it "forgets" the point at where it was executing before migrating. Despite the clear drawbacks of the second type of migration, it is widely supported by most mobile agent platforms because it is easier to implement than strong migration. On the other hand, though strong migration is hard to implement, it is much simpler to use for programming mobile agents than weak migration [23, 18]. The rest of the paper will focus on reactive mobility. Details on proactive mobility in MoviLog are described in [24].

RMF is a novel form of reactive mobility which is based on the assumption that mobility is orthogonal to the rest of the abilities agents may have [25], namely reasoning, reactivity, learning, interaction, and so forth. RMF exploits the conceptual independence among agent abilities at the implementation level by separating agent functionality in 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 deciding about when and where to move. RMF exploits this separation by allowing the programmer to focus his efforts on the stationary functionality. Traditional distributed technologies like RPC and Java RMI simplify application development by hiding the location of components in a network, so that they may interact as if they were located at the same machine. Similarly, RMF hides details and complexities about agent mobility [1], which aims at making mobility easy to use.

Before going into further details we will first define several important concepts. The run-time platform residing at each host that provides support for executing agents is called a *MARlet* (Mobile Agent Resource servlet). A set of MARlets such as all of them know one another conforms a *logical network*. A logical network groups MARlets belonging to the same application or closely related applications. In addition, MARlets can provide resources such as databases, procedures or Web Services to agents.

A mobile agent in MoviLog consists of a sequence of Prolog clauses (code and data) and a possibly empty sequence of *protocols*, which are also Prolog predicates. Protocols represent resources potentially needed by the agent along its lifetime. For example, the behavior of an agent looking for a phrase within a file would be that of applying a string matching algorithm over the file contents. Here, a protocol is required to indicate that the algorithm needs an external resource (the file) to accomplish its task. Indirectly, protocols define the points of an agent's code that may trigger mobility. The idea is that only some particular parts of a mobile agent may produce an *m*-failure, this is, those predicates implementing the agent's functionality described as protocols, whereas the rest of the code has non-mobile behavior. In this way, the developer controls which parts of the agent may cause it to move and which ones not. In our example, each point of the code accessing the file may potentially trigger mobility.

RMF is based on the concept of *m*-failure. An *m*-failure is an attempt for accessing a resource (code, data, services, etc.) that is unavailable at the current agent's location. An *m*-failure can only be caused by a code predicate described as a protocol. When an agent causes an *m*-failure the MARlet at the current location moves the agent to a host with the required resource. Once the agent has migrated to the destination, its execution is resumed. As a result, mobility is transparent to the agent. The *m* in *m*-failures is to distinguish traditional Prolog failures, where no mobility is involved, from failures that may cause an agent to move.

It is worth emphasizing that an agent does not decide neither the time to migrate nor its destination. Migration is triggered by *m*-failures, and then the destination is dynamically selected by the local MARlet by communicating with its peers in the logical network. As a result, even when the agent knows nothing about mobility, RMF can decide when and where to migrate the agent. Indeed, the only information an agent is required to specify about mobility are the code predicates whose failures are to be treated as *m*-failures.

Syntactically, 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 an agent point of view: to let the programmer control the points of an agent's 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 activates when an *m*-failure occurs by searching the logical network for the MARlets providing clauses with the same protocol as the goal that *m*-failed. Protocols enable MARlets to describe the clauses or resources they provide.

A simple example will clarify the ideas introduced so far. We will first show a traditional Prolog program. Then, we will define a protocol to show how the program becomes a mobile agent using RMF. Let us consider the following Prolog code:

```
1 preferred(_, sata, _, RPM, _, _):- RPM >= 7200.
2 ...
3 searchForOffers(CurrentList, FinalList):-
4       hd(Id, Type, Brand, RPM, Capacity, Price),
5       not(member(disk(Id), CurrentList)),
6       preferred(Id, Type, Brand, RPM, Capacity, Price),
7       searchForOffers([disk(Id)|CurrentList], FinalList).
8 searchForOffers(CurrentList, CurrentList).
9 ?- CurrentList=[], searchForOffers(CurrentList, FinalList).
```

, which queries the Prolog database for clauses hd/6 (this is, functor hd and six arguments) representing hard disks and satisfying some users' preferences, these latter represented by preferred/6 (lines 1-2). The result of the program is a list FinalList containing facts of the form disk(Id), where Id is the serial number of the hard disk. The predicate preferred(Id, ..., Price) evaluates to true if the hard disk identified by Id matches a number of preferences over its type, brand, speed, capacity and/or price. Particularly, in our example, the user is interested

Table 1. Three MARlets and their clauses

M_1	M_2	M_3
hd(#123,sata,wd,7200,300,50)	hd(#80,scsi,ibm,15000,73.4,100)	hd(#22,scsi,seagate,15000,300,250)
hd(#23,sata,maxtor,7200,80,30)	hd(#33,sata,samsung,7200,320,55)	hd(#44,sata,panasonic,7200,120,50)
hd(#78,scsi,hp,10000,146,150)	hd(#45,sata,ibm,5200,160,40)	

in retrieving serial ATA disks whose speed is greater or equal than 7200 RPM (line 1). The ?- predicate (line 9) represents the input and output of the program, given by a temporal empty list (CurrentList) and another unbounded list (FinalList) where the search results will be placed.

Basically, the rule of line 3 is in charge of recursively finding and adding hard disks to the temporal result list (line 7) by firstly avoiding duplicates (line 5) and checking that the users' preferences are fulfilled (line 6). Once there are no more items, this rule finally evaluates to false, thus the rule of line 8 is evaluated by Prolog so as to make ?- evaluate to true, which simply copies the contents of the temporal list to the final result list. To further explain the execution of the program we will now consider a Prolog database containing three clauses (column M_1 of Table 1[2]). If we execute the program with those clauses we obtain a list [disk(#123), disk(#23)], stating that the hard disks #123 and #23 match the users' preferences.

The next code implements a modified version of the above program which uses RMF for searching the three MARlets M_1, M_2 and M_3 for hard disks. 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 agent. Basically, the idea behind this code is to trigger mobility upon *m*-failures of predicates hd/6 and hence forcing the program to visit the three MARlets. The modified code is:

```
PROTOCOLS
  protocol(hd, 6).
CLAUSES
  preferred(_, sata, _, RPM, _, _):- RPM >= 7200.
  ...
  ?- CurrentList=[], searchForOffers(CurrentList, FinalList).
```

[2]Prices are hypothetical

As in the previous example, we are searching for hard disks satisfying some preferences. The code behaves the same as the first example up to the point when the program evaluates 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. As a consequence, the RMF run-time will search for MARlets providing clauses hd/6 to migrate the agent 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 agent to M_2, the program continues searching hard disks until no more options are available. At 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 agent. Then, it will be returned to its origin (M_1) by the MARlet M_3. Finally, the result of the execution of the program will be [disk(#123), disk(#23), disk(#33), disk(#44)]. Note that after a successful evaluation of a predicate that *m*-failed an agent does not return automatically to its origin. It returns if it finishes its execution, fails (no more alternatives are available for RMF) or the programmer manually invokes the return primitive, which is provided by MoviLog.

To better understand the example we will show how RMF acts at each step of the execution of the input query. The predicate ?- is similar to executing:

```
hd(Id=#123, Type=sata, Brand=wd, RPM=7200, GB=300, Price=50),
not(member(disk(#123), [])),
preferred(Id=#123, Type=sata, Brand=wd, RPM=7200, GB=300, Price=50),
hd(Id=#23, Type=sata, Brand=maxtor, RPM=7200, GB=80, Price=30),
not(member(disk(#23), [disk(#123)])),
preferred(Id=#23, Type=sata, Brand=maxtor, RPM=7200, GB=80, Price=30),

% An m-failure is triggered
hd(Id=#78, Type=scsi, Brand=hp, RPM=10000, GB=146, Price=150),
not(member(disk(#78), [disk(#23), disk(#123)])),
preferred(Id=#78, Type=scsi, Brand=hp, RPM=10000, GB=146, Price=150),

% Agent is migrated to M2
hd(Id=#80, Type=scsi, Brand=ibm, RPM=15000, GB=73.4, Price=100),
not(member(disk(#80), [disk(#23), disk(#123)])),
preferred(Id=#80, Type=scsi, Brand=ibm, RPM=15000, GB=73.4, Price=100),
...
```

The execution of the first six lines of the code will successfully evaluate the clauses hd(#23, sata, ...) and hd(#123, sata, ...), respectively. The third line will evaluate hd(#78, scsi, ...), but it will fail because preferred will be false (the disk is not serial ATA). At this point, MoviLog will try to find another clause hd, but no more clauses are available at the MARlet M_1. As a consequence, the third evaluation of hd/6 will *m*-fail because it is declared as a protocol. At this point, RMF will move the agent to M_2 and its execution will be resumed. As shown in the example, protocols enable the programmer to delegate mobility decisions on RMF regarding access to resources. Although not illustrated in the example, MoviLog also supports dynamism with respect to the contents of MARlets during mobile agent execution. For instance, if a new clause hd/6 is incorporated to M_1 while the agent is migrating to M_2, the platform detects this situation and updates the execution state of the agent upon its arrival to M_2 so a new alternative for reevaluating hd is taken into account in the future. A similar consistency mechanism is applied to handle deleted clauses, but the algorithm implementing it is rather more complex. See [1] for more details on both of these consistency mechanisms.

In the example, the agent visits all the MARlets containing hard disks. It is worth noting that this behavior is not forced by MoviLog, but by searchForOffers, because it evaluates all the predicates hd to make the query true. In other words, when an individual *m*-failure occurs, RMF moves the agent to one MARlet only, leaving remaining options as *backtracking* points. Because searchForOffers tries to find all the hard disks, it causes three *m*-failures, one at each executing MARlet.

So far we have described an *m*-failure caused by a simple lookup task. This is the simplest *m*-failure a program can cause. In general, an *m*-failure may be caused by any fragment of Prolog code with arbitrary complexity that evaluates to false (fails). This, in turn, may be caused by the absence of a clause, as shown in the example, or other situations such as a calculation whose result is within certain range, a predicate whose execution produces an empty list of facts, an invocation to a Web Service that returns no results [20], etc.

In cases where RMF is not enough to capture and express the mobile behavior of a mobile agent, it is possible to combine RMF with traditional proactive mobility. To this end, MoviLog provides a moveTo(Site) primitive, which causes an agent to migrate to the host represented by Site. Indeed, there are situations where the programmer may know exactly when and where to move an agent, which may yield automatic mobility counterproductive. However, in highly dynamic environments with hosts that enter and leave the network often, proactive mobility is difficult to use and manage. Consider, for example, a mobile agent that has to visit a sequence of hosts and execute some code on each of them. A typical solution using proactive mobility would involve defining the sequence of hosts in some variable of the agent. Then, a simple iteration over the list of hosts would solve the problem. Now, if the hosts are allowed to randomly leave or enter the network, and accept or deny mobile agents, the problem becomes harder because the solution would require some code for updating the itinerary. This is the type of situations where RMF simplifies mobile agent development. Besides, in applications that use mobility for avoiding remote interactions, RMF has also shown significant advantages [1].

Finally, RMF and proactive mobility do not interfere with each other. For example, let us suppose that a mobile agent produces an *m*-failure while evaluating a predicate hd/6 at M_1. As a consequence, the agent is moved to site M_2 (with alternatives M_2, M_3). Then, let us additionally assume that the mobile agent executes an explicit moveTo(M_3) when executing at M_2. In case hd/6 is reevaluated the agent will move again to M_2. Indeed, RMF remembers the sites visited by the agent for reevaluating the goal that caused an *m*-failure by keeping a list of sites that may be visited to look for more clauses if the goal requires further reevaluations.

Up to now we have described how RMF handles mobility automatically. The next section explains the policy support for adapting RMF according to the characteristics of mobile agents.

3. Resource access policies and mobility policies

Despite RMF has shown positive results [1], performance problems may arise due to the lack of information RMF has about the application being executed. For example, consider an agent that causes an *m*-failure, and then RMF migrates the agent to a remote MARlet. Once there, the agent just accesses a clause such as hd available locally and then returns to its origin. The problem here is that the two migrations of the agent are more expensive in bandwidth and time than the cost of copying the clause needed to solve the failure from the remote MARlet to the current agent's location.

Something similar occurs when, after an *m*-failure, a clause is available at several MARlets. Then, RMF has to decide where to migrate the executing agent. If blindly decided, the agent may end running on a heavily loaded MARlet, traveling through several slow network links, or even worst, executing on a site that charges for CPU usage. Up to now, these situations were avoided by the programmer by combining RMF with proactive mobility, this is, by using moveTo(Site) to instruct the agent to migrate to Site. However, by doing so one of the main benefits of RMF, namely the separation between agent functionality and mobility, is lost.

From now on, the paper will present a method for adapting RMF to different application requirements while maintaining the separation of agent functionality and mobility. The basic idea is to allow the agent developer to specify *policies*, which govern choices in the behavior of RMF. In other words, policies are rules provided by the programmer to adapt RMF to his requirements.

The contribution of this mechanism is to show that it is possible to automate mobility decisions. Unlike the original RMF where mobility is fuly automatic [1], this work aims at providing flexibility on how mobility decisions are made. As we will show, the main consequences of the approach are:

- Separation of concerns: agent functionality does not get mixed with code for handling mobility. This makes agent code easier to develop, maintain and understand. In addition, it is possible to dynamically change mobility strategies. For an in-depth discussion and evaluation of the benefits of separation of concerns in multi-agent systems see [21].

- Performance: mobility has advantages with respect to performance in those cases where remote interactions are frequent or expensive. However, mobility has been hard to use. The approach presented in this paper makes mobility easier to use and exploit compared to traditional proactive mobility. We do not claim that by using RMF in conjunction with policies we obtain better performance, but that by using these supports it is easier to build mobile agents that take benefit from mobility. In principle, performance comes from the fact that most of the code for handling mobility is provided by the run-time when using RMF and policies. In opposition, the code for handling mobility is embedded into the agent code under proactive mobility. As a result, RMF-enabled mobile agents have to carry less code, use less network bandwidth and move faster.

Despite the rest of the paper will describe policies in the context of MoviLog, these ideas are applicable to other programming languages with support for strong mobility. For example, we have made some interesting progress towards supporting RMF and policies in the Java language [26]. However, this paper focuses on explaining and evaluating the policy mechanism based on MoviLog and hence the Prolog programming language, because MoviLog represent at present the most stable prototype implementation of our ideas.

From now on, we will distinguish between two types of policies:

- *resource access policies*: in the previous example, *m*-failures always cause the migration of the executing agent. However, *m*-failures can be treated either by moving to other MARlets of the network or by copying clauses from other MARlets to the local one, depending on several factors such as network traffic, resource usage, etc. Resource access policies are rules for deciding whether to migrate an agent or fetch a resource located at a remote MARlet.

- *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. Mobility policies define the destination for an agent when more than one destination is available after an *m*-failure.

The next sections describe the two types of policies in detail.

3.1. Resource access policies

Resource access policies (RAP) are used for deciding whether to migrate an agent or fetch required resources from remote MARlets. For example, a simple RAP is to migrate an agent if the network traffic produced by migrating the agent is less than the estimated traffic for fetching some Prolog clauses. An RAP is a rule with four parts, *protocol*, *condition*, *actionT* and *actionF*, where protocol is a pair name/arity for associating the RAP with a protocol, *condition* is user-defined code, and *actionT* and *actionF* may take the following values:

- *move*: migrates the agent to a MARlet offering the required resource.

- *fetch*: transfers one instance of the required resource from a remote MARlet.
- *fetchAll*: fetches all instances of the required resource from a remote MARlet.

When an *m*-failure occurs, the condition of the RAP associated with the protocol of the goal that *m*-failed is evaluated. *ActionT* is executed if *condition* holds, otherwise *ActionF* is executed.

In the example described in Section 2. we could use a resource access policy to decide whether to migrate the agent or fetch all the clauses hd/6 from a remote MARlet instead (recall that in Prolog capitalized arguments are variables whereas the rest are constants):

```
1  PROTOCOLS
2    protocol(hd, 6).
3    accessPolicy(hd,
4             6,
5             (agentSize(T1), estTrFAll(hd, 6, M, T2), T1 < T2),
6             move,
7             fetchAll).
8  CLAUSES
9    preferred(_, scsi, _, _, _, _).
10   ...
11   ?- CurrentList=[], searchForOffers(CurrentList, FinalList).
```

accessPolicy (lines 3-7) states that in case of an *m*-failure the agent migrates (line 6) if the network traffic $T1$ required for migrating the agent *A*, denoted agentSize(T1), is less than the estimated network traffic $T2$ required for fetching all the clauses hd/6 from MARlet *M*, denoted estTrFAll(hd, 6, M, T2). This condition is encapsulated in line 5. Otherwise, all the clauses hd/6 are fetched from *M* (line 7). Note that *M* is instantiated by RMF with the next site to visit.

The condition of the previous example uses two simple built-in rules provided by MoviLog for estimating network traffic: agentSize and estTrFAll. The first one just returns the agent size measured in bytes. The second one queries *M* for the size of one of its clauses hd/6 and how many of them are available, and then multiplies both numbers. Note that the resulting value is just an estimation, which may be accurate or not depending on the data. In addition, the programmer is free to specify any estimation mechanism by using custom Prolog predicates. To this end, MoviLog offers an extensible binding that allows users to implement library-like functions in other languages (currently Java) and wrap such functions as Prolog predicates, thus they are accessible to mobile agents.

Let us consider a variation of the previous example where a small subset of the sites are connected to the rest of the network by using wireless links. These links are unreliable and thus force mobile agents to perform several retries to fetch data, where the overhead is inversely proportional to the link quality. A policy may be used for evaluating:

- the cost of migrating through the wireless links by taking into account the agent size plus the extra traffic involved in retrying the agent transfer.
- the cost of fetching the list of hard disks from the destination plus the extra traffic involved in retrying due to errors.

Then, the code implementing this variant of the application would be:

```
PROTOCOLS
  protocol(hd, 6).
  accessPolicy( hd, 6,
    ( wifi(M) ->
      ( agentSize(T1), linkQuality(M, Q), T1E is T1*(2-Q),
        estTrFAll(hd, 6, M, T2), T2E is T2*(2-Q)
        T1E < T2E )
        ) ;
      ( agentSize(T3),
        estTrFAll(hd, 6, M, T4),
        T3 < T4 ) ),
     move, fetchAll ) .
CLAUSES
  preferred(_, scsi, _, _, _, _).
  ...
  ?- CurrentList=[], searchForOffers(CurrentList, FinalList).
```

Here, the most interesting part of the previous example is the definition of the RAP. It first determines whether the network link to the next destination MARlet M is wireless. In that case, the RAP estimates the traffic caused by migrating the agent and fetching the clauses from M plus the overhead of retransmitting. Otherwise, it just uses the same policy as the previous example.

The previous example shows just a glimpse of the flexibility of RAPs. Note that in the example, the predicate wifi(M) and lnkQlty(M,Q) are defined by the language, but nothing prevents the programmer to use predicates that depend on the internal or external state of the agent. In addition, the evaluation of a policy may also do something besides performing simple calculations. For example, alter the agent behavior, invoke a Web Service, or anything that can be written in Prolog.

3.2. Mobility policies

When an *m*-failure occurs, there may be several alternatives for solving it. For example, a resource may be offered by two or more MARlets. In this sense, a mobility policy (MP) selects the next destination for moving an agent or the next source for fetching resources based on some user defined metric such as remote CPU load or network link speed.

In addition, an MP can build an itinerary for an agent if necessary. For example, let us suppose an agent that *m*-fails when evaluating a goal a(X). There are two MARlets M_1 and M_2 offering resources a/1. The agent moves to M_1, but after several tries when reevaluating a(X) the agent fails again. Intuitively, an alternative would be to try at M_2, but M_2 may not offer resources a/1 anymore or a new MARlet M_3 with plenty of RAM and CPU power may join the network. MPs not only select the next destination for moving agents, but also maintain an up-to-date itinerary for them.

As in the case of RAP policies, the basis of this mechanism consists on associating an MP to each protocol. The policy is used for selecting the next destination for reevaluating a goal given a set of MARlets offering the same protocol for the goal. For example, in a CPU bound application, it may 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 example of the previous subsection, we could now 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).
  mobilityPolicy(hd, 6, shortestMoveTimePolicy).
  ...
```

At the implementation level, this policy uses IP ICMP echo requests to determine the network delay between hosts. Examples of other policies are:

- cpuLoadPolicy: obtains the CPU load of the MARlets offering the specified protocol and then sorts them in ascending order. The resulting itinerary is updated if the goal is further reevaluated.
- freeMemoryPolicy: obtains the available memory for executing agents of the MARlets offering the specified protocol and then sorts them in descending order. The resulting itinerary is updated if the goal is further reevaluated.
- randomOrderPolicy: random ordering.

In all cases, MARlets that leave or join the network are taken into account [1]. For efficiency reasons, MARlets communicate among them by using a multicast peer to peer protocol specially designed for RMF. It is out of the scope of this paper to discuss the internals of this protocol. For further details on it, please refer to [27].

Up to now, in the examples shown, itineraries are built by RMF dynamically. However, there are situations where the set of hosts is predetermined. The next example shows the code of an agent that finds documents containing the string "some string". Each site of the network may contain several documents. This is represented by a predicate documents(Docs) (line 14), where Docs is the list of documents available at the current executing site.

The code indirectly assigns a static itinerary for evaluating documents(Docs) by attaching a mobility policy (line 3) to the protocol document/1 (line 2). Clearly, the declared protocol causes the predicate documents(Docs) to be subject to the control of RMF, while the attached policy determines the itinerary for evaluating the predicate. The code [M_1, M_2, [M_3->M_4;M_5]] (line 3) represents the itinerary "go to M_1, M_2 and M_3 in sequence. If M_3 is unavailable, go to M_5; otherwise go to M_4". Basically, each item of the itinerary is consumed every time the predicate documents(Docs) is reevaluated.

```
1  PROTOCOLS
2    protocol(document, 1).
3    mobilityPolicy(document, 1, _, [M1, M2, [M3->M4; M5]]).
4  CLAUSES
5    % findDocuments(Docs, _, String, MatchingDocs)
6    % MatchingDocs is a subset of the documents in Docs
7    % that contains String
8    findDocuments([], MatchingDocs, _, MatchingDocs):-!.
9    findDocuments([Doc,Docs], MatchingAux, String, MatchingDocs):-
```

```
10      documentContains(Doc, String),
11      findDocuments(Docs, [Doc|MatchingAux], String, MatchingDocs).
12    % Finds all the documents containing some string
13    findDocuments(_) :-
14      document(Docs),
15      findDocuments(Docs, [], some string, MatchingDocs),
16      currentSite(S),
17      assert(matching(S, MatchingDocs)), fail.
18    findDocument(L) :-
19      collectDocuments([], L), !.
20    % Collects the facts matching(site, [doc1, doc2, ...]) into a list
21    collectDocuments(Aux, MatchingDocs) :-
22      matching(S, Docs), !,
23      collectDocuments([[S,Docs]|Aux], Docs).
24      collectDocuments(MatchingDocs, MatchingDocs).
25    ?- findDocuments(FinalList).
```

The most interesting part of the above code is the clause findDocuments. It first evaluates document(Docs) to obtain the list of documents hosted at the current site. Then, the code iterates through the list of documents to find those that contain "some string". When this algorithm finishes, the code adds to the agent's state a fact matching with the name of the current site and the list of documents found that passed the contents filter. Afterwards, the code forces the reevaluation of documents(Docs) because of the fail predicate at the end of line 17. The idea is that by reevaluating that predicate, RMF will migrate the agent to the next site of the attached itinerary.

The syntax of a mobility policy is mobilityPolicy(functor, arguments, ordering, itinerary). The first two parts (functor and arguments) determine the affected protocol, ordering represents a mechanism for sorting the hosts of the itinerary ("_" is a do nothing sorting strategy) and itinerary specifies a static itinerary for the given protocol. Note that by specifying an itinerary the programmer overrides the default behavior of RMF for dynamically building itineraries. In addition, a static itinerary may be prefixed by "+" for instructing RMF to update the itinerary when sites join or leave the network. In this case ordering is taken into account for updating the itinerary with new sites.

It is worth noting that both protocols and policies can be manipulated at run-time by the agent. Indeed, protocols can be modified, RAPs and MPs can be added or removed as any other Prolog fact. The only limitation is that they cannot be removed while they are in use to solve an *m*-failure. All in all, by extending RMF with policies we increased its flexibility while maintaining its advantages regarding ease of programming. In addition, the separation of agent behavior and mobility code is increased. This, in turn, has a positive effect on maintainability, modifiability and testability of agent code, which are desirable quality attributes for any kind of software. By using RMF and policies, mobile agents are developed in a two-step way [21]: a stationary version of the agent containing the pure functional code is first derived, which is then furnished with the code in charge of performing mobility.

4. Experimental results

The next subsections reports some experimental results obtained with two applications: a distributed solver and a computer components searcher. The goal of the experiments is to evaluate the benefits of the extended RMF support and policies.

4.1. A distributed solver

We developed a distributed mathematical expression solver by using proactive mobility (PM), RMF, RMF+policies, and message passing without mobility (traditional client/server or non-mobile) and compared the four approaches. In this experiment, even for the variants not relying on mobility behavior, we used MoviLog in order to avoid language differences. The application consists of two types of agents:

- *server*: a server is a non-mobile agent that provides services for solving binary arithmetic operations such as +, -, /, or *. Servers cannot handle compound operations, but are only able to solve a single type of arithmetic operation. In addition, each server is statically assigned to a specific host of the network.

- *client*: is an agent 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, the expression:

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

Table 2. Mathematical expressions used in the tests

Test case	Mathematical expression
1	$(10 - 7) + \frac{9*2}{3-1}$
2	$((3 * 1) + (3 - 1))!$
3	$(\sum_{i=1}^{6} i + \sum_{i=1}^{4} i)^3$
4	$35 + \frac{average(100,2,84,5,15,1,20,4,53)}{4}$
5, 6 and 7	$5^1 + 2^2 + 3^3 + 4^4 + 5^4 + 2^3 + 3^2 + 4^1 + 4^3 + 2^4$

can be rewritten as a number of binary operations: $(((2*2/(3-1))+10)-7$. In this way, these binary operations can be individually solved by servers.

Since servers and clients may reside in different sites, and remote communications between agents are not allowed (except in the non-mobile solution, in which the client made requests to servers by employing remote synchronous messaging provided by MoviLog), clients may migrate to ask for the resolution of a binary operation. When two or more servers located at different sites are able to solve the same required operation, clients try to balance the CPU load of the network hosts by migrating to the least loaded site. In the variants not based on RMF, the capabilities for solving operations of the servers were known in advance, whereas in the RMF-based solutions this was of course determined by the run-time in charge of handling *m*-failures.

Seven different test cases were implemented by using the four different mobility mechanisms, this is, PM, RMF, RMF+policies, and client/server or non-mobile. The expressions employed in the tests are shown in Table 2. Then, the seven test cases and their four implementations were run on a 100 Mbps LAN with four PCs using Java 6.0 and Windows XP. Four server agents capable of solving different binary operations were deployed on each computer. In addition, the computers also ran a process that generated random CPU load to provide a more realistic and challenging scenario.

The test cases consisted on a client agent that was ordered to solve a mathematical expression by using 16 server agents distributed across the four com-

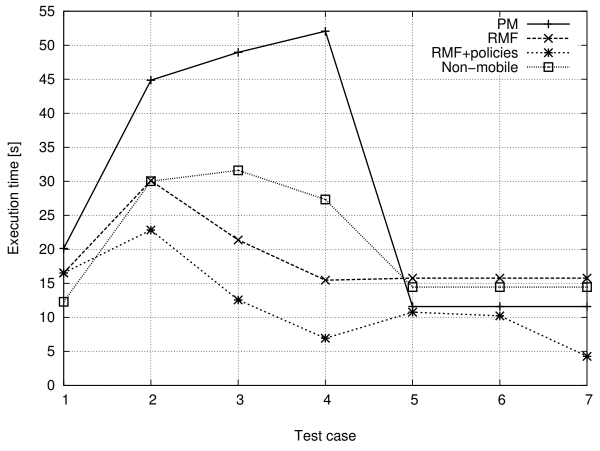

Figure 1. Execution time

puters. The first five test cases varied in the complexity of the mathematical expression. On the other hand, the test case 5 used a cache at each server for storing partial calculations. Test cases 6 and 7 used the same expression as 5, but the solution with RMF+policies used two different RAPs heuristics for fetching partial solutions from the cache or moving the agent. Furthermore, for RMF+policies, in all the test cases we used an MP to migrate agents to sites with low CPU load. In the first five test cases RAPs were not used. Figures 1 and 2 show the average running time and network traffic for 10 executions of each test case. The standard deviation of the results was less than 5%.

For each test case $i \in 1,2,3,4,5,6,7$ and each implementation $j \in$ PM, RMF, RMF+policies, Non-mobile, we calculated the improvement ratios with respect to execution time and traffic as follows:

$$timeRatio_{i,j} = 1 - \frac{time_{i,j}}{\sum_j time_{i,j}/4}$$

$$trafficRatio_{i,j} = 1 - \frac{traffic_{i,j}}{\sum_j traffic_{i,j}/4}$$

For a given implementation j and test case i, $timeRatio_{i,j}$ (or $trafficRatio_{i,j}$) can take the following values:

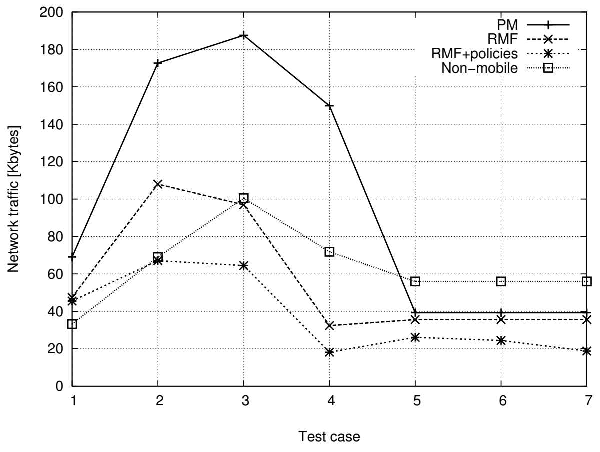

Figure 2. Network traffic

- < 0 when $time_{i,j}$ (or $traffic_{i,j}$) is greater than the average time (or traffic) for the test case i. This means that the performance in terms of time (or traffic) of the implementation j is below average for the test case i. In this case, the smaller the ratio, the worst the performance (or network usage).

- $= 0$ when $time_{i,j}$ (or $traffic_{i,j}$) is equals to the average time (or traffic) for the test case i. This means that the performance in terms of time (or traffic) of the implementation j is equals to the average for the test case i.

- > 0 when $time_{i,j}$ (or $traffic_{i,j}$) is less than the average time (or traffic) for the test case i. This means that the performance in terms of time (or traffic) of the implementation j is above average for the test case i. In this case, the larger the ratio, the better the performance (or network usage).

Figures 3 and 4 show the execution time and traffic improvement ratios for the seven test cases. As shown in the Figures, proactive mobility performed rather poorly. This was mainly caused by the size of the agents, which impacted on the data transferred upon each invocation to the moveTo mobility primitive. They were 23% bigger than the agents built with RMF+policies. This difference in size was caused by the code for handling mobility, which was negligible in the implementation using RMF.

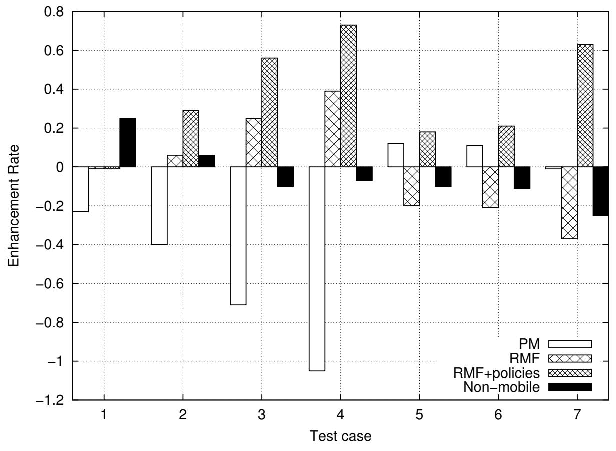

Figure 3. Improvement ratio (execution time)

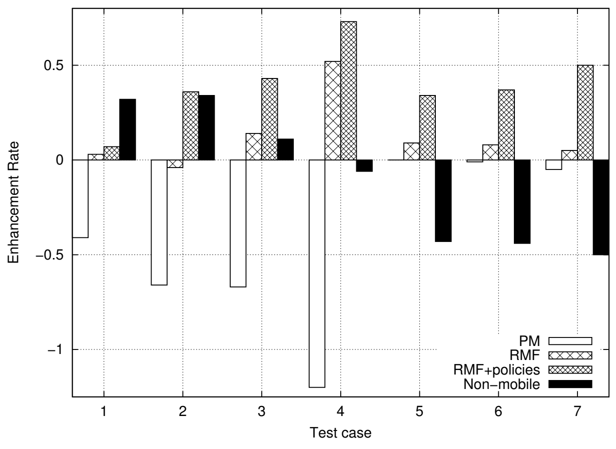

Figure 4. Improvement ratio (network traffic)

On the other hand, the performance of the non-mobile solution was in the middle between RMF and proactive mobility. The non-mobile solution had serious problems when computing expressions with several (and different) mathematical operators as clients and servers interacted often and were located at different sites. In those cases, mobility used less network traffic because remote interactions were minimized by migrating clients. RMF performed well in test cases 1 to 4, but rather badly in cases 5 to 7 because RMF moved the agent more times than necessary. Finally, RMF+policies outperformed the other solutions in 6 cases, which shows the usefulness of the policy mechanism for improving the efficiency of RMF-enabled mobile agents.

It is worth noting that the size of the agent implemented with RMF+policies was 23% smaller than the one implemented with proactive mobility. This shows that RMF+policies, at least for this experiment, required less code to implement the applications. Furthermore, less code implied that migration was faster and less network bandwidth was used when moving RMF-based agents in most cases. In addition, the solution using RMF+policies was 5% bigger than the smaller solution (RMF) because of the extra code for the policies. Note that in general policies are compact. As mentioned in past sections, for more complex policies, MoviLog provides a Java binding and API for defining policies by wrapping existing Java code. With these APIs it is possible to extend MoviLog with shared user-defined policies. As a consequence, agent size can be further reduced.

The importance of these results are twofold. On one hand, they show that by using RMF, agents are smaller than proactive mobile agents because of the simplification of the code for handling mobility. However, in some situations the price for easier development is loss of efficiency. On the other hand, the results suggest that agents using RMF+policies are faster and use network resources more efficiently than traditional proactive mobile agents and RMF. All in all, by extending RMF with policies we obtained better results than with RMF, proactive mobility and no mobility.

4.2. A computer components searcher

We developed another application consisting of an agent for distributed search of computer parts. We deployed a MoviLog network comprising the above four sites with data and clauses representing hardware supplies. Basically, each site contained information about specific computer components (disk, motherboard, memory, processor, monitor and keyboard). Furthermore, two variants of the searcher agent were used as test cases:

- test case 1 (search with preferences): an agent was asked to search for computer parts satisfying a number of constraints over its features. In this problem, the agent had to migrate several times until all the parts required to build a computer were found.

- test case 2 (search the cheapest computer): this was similar to the previous case, but the goal here was to build an entire computer by finding the cheapest computer parts. Hence, the agent had to check the price of each part on all the sites before "buying" it.

For each test case, we implemented five solutions:

- Proactive mobility by using MoviLog.

- Proactive mobility by using Jinni [28], a Prolog-based language with extensions for mobility.

- RMF.

- RMF with a MP that ordered sites according to the latency of the network links.

- RMF with the same MP as the previous solution and a RAP that decided whether to migrate the agent or fetch a computer part according to the estimated network traffic.

The implementations were run on the same network described in subsection 4.1.. Figures 7 and 8 show the average execution time and network traffic for 10 executions of each implementation of the application. Moreover, Figures 5 and 6 show the execution time and traffic improvement ratios for the two test cases. The ratios were calculated as explained in the previous subsection.

Once again, reactive mobility performed much better than its proactive counterpart. From Figures 7 and 5, we can see how the usage of MPs first, and the combination of MPs and RAPs later, consistently improved the default behavior

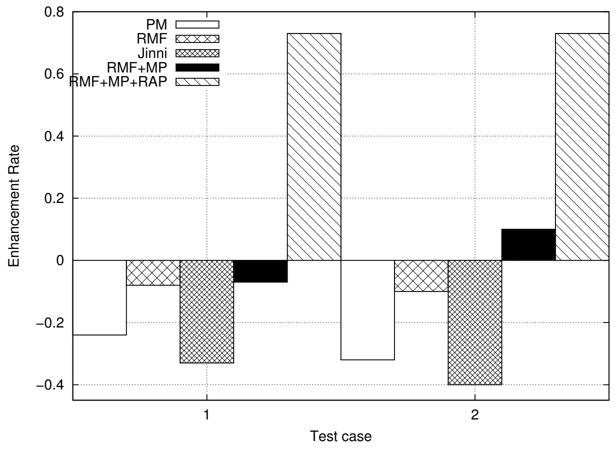

Figure 5. Improvement ratio (execution time)

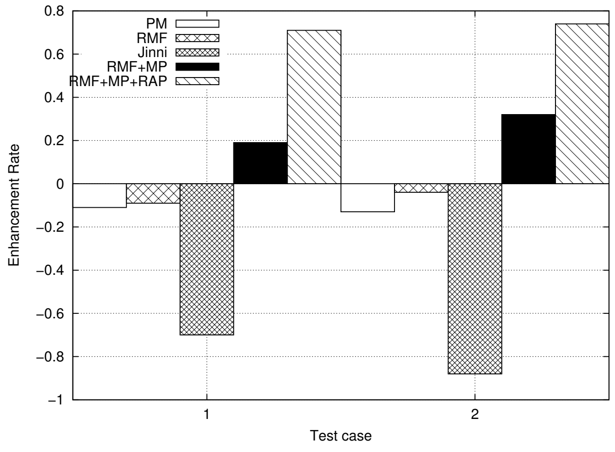

Figure 6. Improvement ratio (network traffic)

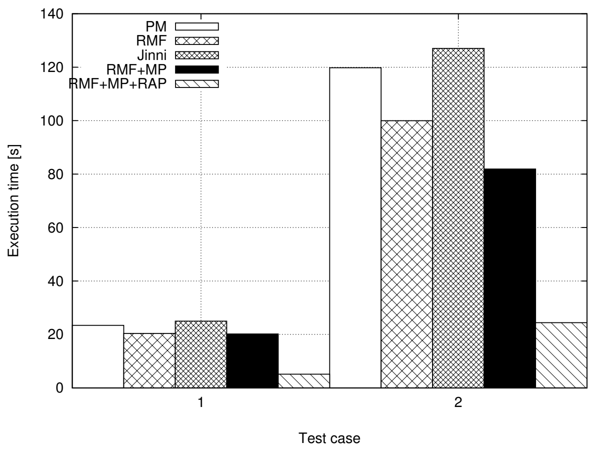

Figure 7. Execution time

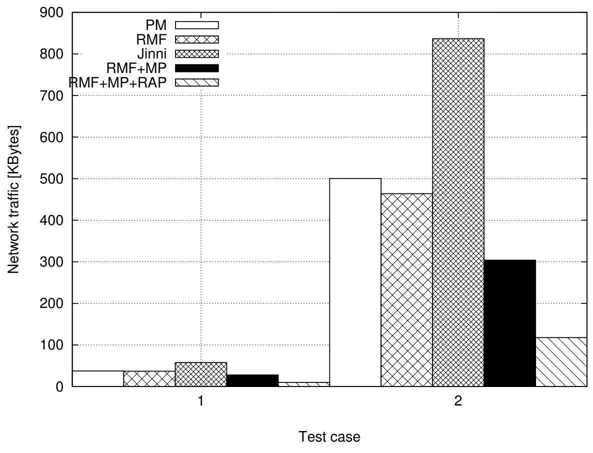

Figure 8. Network traffic

of RMF. A similar situation occurred for the overall network traffic generated during agent execution, as shown in Figure 8.

Finally, the Jinni implementations of the two test cases used more network resources than RMF and proactive mobility. In spite of this fact, we originally decided to use Jinni as it is a programming language for mobile agents that is very close to our goals, this is, simplifying the development of mobile agent-based applications. On the other hand, like MoviLog, Jinni is based on Prolog, which allowed us to perform a fair comparison between the applications in terms of the necessary codes lines to implement the various benchmarks.

4.3. Analysis of the results

In the experiments, RMF+policies consistently yielded speed and traffic improvements. These results were a direct consequence of the reduction in code size, because most of the code for managing mobility, except policies, is provided by the run-time. In opposition, the implementations with proactive mobility (MoviLog and Jinni) included many lines of code for handling mobility. Besides improving performance, policies allowed the programmer involved in the experiment to separate agent functionality from mobility-related code. This is by itself a strong advantage with respect to the discussed related approaches.

It is worth noting that the usage of policies does not directly imply that performance is increased. The bigger the ratio:

$$\frac{size\,of\ clauses\,section}{size\,of\ policies}$$

, the better the performance with respect to existing approaches. Indeed, the benefits in terms of speed and network usage come from the fact that agents are smaller because some of the code for handling mobility is pushed down to the run-time support. For situations where policies become complex and require many lines of code, MoviLog provides two APIs (Prolog plus Java) for incorporating policies to the run-time platform. In this way, these policies do not have to be carried by the agent and therefore do not require extra network bandwidth. In addition, “exported” policies can be reused across several applications.

Last but not least, MARlets avoid moving agent policies multiple times by using a simple caching technique. This is possible because the agent code is split in two parts and policies usually do not change. In Jinni, for example, this is not possible, because the agent code usually changes during run-time. This, in turn, is a result of the lack of separation between code and data.

5. Related work

There are many tools for supporting mobile agent development, however, most of them only provide rudimentary mechanisms for handling agent migration based on proactive mobility or weak migration. As a consequence, programmers are responsible for dealing with mobility, this is, providing code for determining when and where to migrate an agent in order to access to non-local resources. Then, developing mobile code demands more programming effort. On the other hand, the spectrum of tools not relying on any technique for separation of concerns force developers to mix application logic with mobility-related instructions.

Some of the earliest efforts to provide high level programming models to facilitate the development of mobile applications are Concordia [29], Aglets [30] and Ajanta [2]. Concordia was one of the first 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 proactive but a simple approach for managing mobility and reducing programming effort. However, Concordia uses a weak migration mechanism. Consequently, the programmer has to adopt a rather difficult event driven approach for programming mobility [23]. A similar problem arises with Aglets as well as Ajanta. Moreover, Jinni 2004 [28] is a tool for programming Prolog-based mobile agents. Jinni 2004 has been specially designed to support orthogonal language constructs, namely program composition and code reuse mechanisms, Prolog-based inference mechanisms, execution of multiple independent goals in both co-routining and multithreaded execution mode, agent coordination and communication, and client-server remote calls. Developers can use Jinni 2004 to exploit proactive strong mobility by embedding mobility primitives in their agents' code. In contrast, MoviLog is based on a strong mobility mechanism that is quite transparent and is offered in the context of an easy-to-use programming model based on reactive mobility.

Aglets introduced the concept of mobility patterns, this is, recurring solutions that appear multiple times in mobile systems. Aglets defines a number of predefined patterns and provides implementations for them, such as *meeting* (several agents are required to meet at a specific host of a network in order to exchange messages), or *slave* (an agent carries a message from one agent to another). Despite the benefits in terms of development effort these patterns provide, developers are still in charge of programmatically handling most mobility decisions. Besides, the weak migration mechanism used by Aglets has the same problems as that of Concordia. Ajanta takes the concept of itineraries a step further by providing migration patterns, this is, abstract migrations paths for agents. 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 defined at development time instead of at run-time as RMF does. Then, Ajanta has problems targeting highly dynamic networks such as the Internet or Computational Grids.

Other approaches such as MAGE [31], MobiPADS [32] and Poema [33] support dynamic reconfiguration for mobility-based distributed applications. MAGE organize applications as a number of distributed components that interact among them. Developers define, by employing an event-based scripting language, distribution strategies at the inter-component level that can be changed at run-time. Then, the code for handling mobility is mostly separated from the application logic. Furthermore, MobiPADS uses a reflective middleware with meta-objects for reconfiguring object-oriented applications. MAGE and MobiPADS require however many modifications in the application code. For example, with MAGE, the programmer has to change the code in the points where components interact. Both tools only support mobility based on weak migration. Indeed, these approaches do not aim at being easy to use but at offering some separation between application code and location aware code. Moreover, Poema [33] differs from these two works in its ability to specify mobility strategies at a higher level of abstraction and to support their seamless modification even during application execution. Strategies are specified separated from the application code by using an ad-hoc policy specification language. Precisely, RMF+policies differs from MAGE and MobiPADS in that, like Poema, it supports true separation of concerns between the functional code of agents and the mobility-related code, and its support for strong migration. However, unlike Poema, our work allows developers to both specify policies and implement agent behavior by using the same language, which provides uniformity.

Some Java-based platforms support strong migration by either extending the Java Virtual Machine (JVM), or automatically transforming ordinary Java source codes to their mobile counterpart. For example, NOMADS [34] uses a special virtual machine called AromaVM, which is a from scratch implementation of the Java Virtual Machine capable of capturing and transferring thread execution state. This idea is still around with programming tools like Mobile JikesRVM [35], which extends a JVM or their own called JikesRVM with mobility functionality. However, these approaches are brittle since interoperability at the JVM level is compromised. X-KLAIM [36] modify Java classes for capturing and reestablishing the state of running threads by relying on source-to-source translation. As an advantage, X-KLAIM does not require an extended virtual machine. However, the modified code is very hard to read and debug, which compromises code maintainability.

Moreover, there is a substantial body of works regarding Java platforms for developing computing intensive parallel applications. Such efforts have born as a consequence of the increasing availability of computational power such as multi-core machines, clusters, Grid and Clouds. As an example, ProActive [37] allows developers to construct applications by composing mobile entities called *active objects*. Active objects serve method calls from other (remote) active or regular objects, and viceversa. Active object creation, lookup and mobility are programmatically performed via API function calls, which requires knowledge on the ProActive API. Besides, the tool is based on weak mobility. Moreover, JavaSymphony [38] and Babylon [39] are two Java platforms featuring semi-automatic execution model that transparently deals with migration, parallelism and load balancing of thread-based applications. Programmers can control such features by means of API primitives in the application code for optimization purposes. However, JavaSymphony and Babylon do not prescribe any mechanism for separating the optimization code from the pure functional code of an application. Finally, mobility for complex systems is not only circumscribed to Java, as evidenced by other mobile agent platforms implemented in compiled languages. A representative example is Mobile-C [40], which supports migration of both C and C++ codes. Unfortunately, Mobile-C is also based on weak mobility, thus it is not able to transfer and restore the execution state of mobile agents.

All in all, the main advantage of our approach is its capability for automatically handling mobility, including dynamic itineraries, while being more efficient than traditional proactive mobile systems. In addition, RMF+policies requires less overall code than related approaches. As a consequence, agents are smaller, use less network bandwidth to migrate, and are easier to develop, understand and maintain. Another important advantage of our approach is the separation of agent behavior and mobility, which improves the quality of the resulting agent code from a software engineering perspective. Finally, MoviLog runs on existing versions of the JVM, thus ensuring JVM-level interoperability.

6. Conclusion

In this paper we have described an extension of RMF aimed at improving its efficiency and flexibility by allowing the developer to adapt the mechanisms used by RMF for performing mobility decisions. Conceptually, our approach shows that mobility can be supported by the middleware so that mobility can be used as easily as any other technology for distributed systems such as RPC, RMI or CORBA, without sacrificing the above aspects. The extension presented in this paper has been implemented and compared with client/server, proactive mobility and RMF. The experimental results are very encouraging since they show important gains in performance and reduced network bandwidth.

By employing our extended approach, the performance of applications is competitive compared to the one achieved with RMF or traditional forms of mobility. Besides, mobile agent development is simplified and the core agent functionality is separated from code for handling mobility. However, despite the obtained results, we are planning to use our new support to experiment with more elaborated applications and other scenarios. A recent successful experience that could be used as a starting point for more experimentation is Chronos [41], a mobile-based meeting scheduling system that is entirely developed in MoviLog.

Another issue with RMF is its requirement for efficient group communication services or multi-cast in order for a MARlet to keep track of the resources other MARlets have. This is difficult to achieve as most approaches for multicast either require special network routers or are specially designed for multimedia content. In addition, most of them have problems handling more than 1000 hosts and multiple senders, which might restrict the applicability of our platform. To efficiently support reactive mobility in large scale deployments we have developed a novel group communication mechanism named GMAC taking into account the special requirements of RMF [27].

We are also 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 effectively solve the *m*-failure. In this context, there are three alternatives: 1) move the agent where the resource is located, 2) move the resource where the agent is located or 3) use some remote invocation mechanism for accessing 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, we have already obtained some interesting results [20].

Despite the description of RMF and policies were tied to MoviLog, the concepts can be applied to other programming languages as well. Indeed, we have developed a prototype Java-based platform with support for RMF and policies [26]. The platform does not require an extended JVM. Basically, this approach has been historically followed by many mobile agents platforms as Java does not allow by default to obtain the execution state of applications and their threads, which is precisely mandatory to support strong migration. Our prototype platform relies on run-time techniques for bytecode modification (the binary code generated by the Java compiler) to replicate the JVM-level execution state of threads as application-level stacks on the Java heap, which are accessed upon migrating an agent. In this way, neither the JVM nor the source codes of applications are affected.

Finally, we are investigating how to adapt MoviLog in order to make it FIPA compliant, a well-established international association that delivers standard specifications that support the materialization of interoperable agent platforms. We aim at allowing MoviLog agents to interact with FIPA enabled platforms and multi-agent systems. As a starting point, we will base our study on a recent FIPA-inspired proposal for interoperable *mobile* agent techniques and platforms [42].

Acknowledgments

We acknowledge the financial support provided by ANPCyT through grants PAE-PICT 2007-02311 and PAE-PICT 2007-02312.

References

[1] Alejandro Zunino, Marcelo Campo, and Cristian Mateos. Reactive mobility by failure: When fail means move. *Information Systems Frontiers - Special Issue on Mobile Computing and Communications*, 7(2):141–154, 2005.

[2] A. Tripathi, N. Karnik, T. Ahmed, R. Singh, A. Prakash, V. Kakani, M. Vora, and M. Pathak. Design of the Ajanta system for mobile agent programming. *Journal of Systems and Software*, 62(2):123–140, 2002.

[3] Robert Gray, George Cybenko, David Kotz, and Daniela Rus. Mobile agents: Motivations and state of the art. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*. AAAI Press/MIT Press, 2001.

[4] S. Manvi and M. Kakkasageri. Multicast routing in mobile ad hoc networks by using a multiagent system. *Information Sciences*, 178(6):1611–1628, 2008.

[5] Vijay Verma, Ramesh Joshi, Bin Xie, and Dharma Agrawal. Combating the bloated state problem in mobile agents based network monitoring applications. *Computer Networks*, 52(17):3218–3228, 2008.

[6] Irene Sygkouna and Miltiades Anagnostou. Efficient information retrieval using mobile agents. In *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, pages 1241–1242, New York, NY, USA, 2005. ACM Press.

[7] Subrata Kumar Das, Kurt Shuster, Curt Wu, and Igor Levit. Mobile agents for distributed and heterogeneous information retrieval. *Information Retrieval*, 8(3):383–416, 2005.

[8] Tainchi Lu and Chinghao Hsu. Mobile agents for information retrieval in hybrid simulation environment. *Journal of Network and Computer Applications*, 30(1):244–264, 2007.

[9] Luminita Vasiu and Qusay H. Mahmoud. Mobile agents in wireless devices. *Computer*, 37(2):104–105, February 2004.

[10] Mustafa Adacal and Ayse B. Benner. Mobile Web Services: An new agent-based framework. *IEEE Internet Computing*, 10(3):58–65, May/June 2006.

[11] J. Nichols, H. Demirkan, and M. Goul. Autonomic workflow execution in the Grid. *IEEE Transactions on Systems, Man and Cybernetics - Part C: Applications & Reviews*, 36(3):353–364, May 2006.

[12] Munehiro Fukuda, Koichi Kashiwagi, and Shinya Kobayashi. AgentTeamwork: Coordinating Grid-Computing jobs with mobile agents. *Applied Intelligence - Special Issue on Agent-Based Grid Computing*, 25(2):181–198, 2006.

[13] Yuhong Feng, Wentong Cai, and Jiannong Cao. Dynamic partner identification in mobile agent-based distributed job workflow execution. *Journal of Parallel and Distributed Computing*, 67(11):1137–1154, 2007.

[14] David Kotz, Robert Gray, and Daniela Rus. Future directions for mobile agent research. *IEEE Distributed Systems Online*, 3(8), August 2002.

[15] P. Fradet, V. Issarny, and S. Rouvrais. Analyzing non-functional properties of mobile agents. In *3rd International Conference on Fundamental Approaches to Software Engineering (FASE'00) - European Joint Conferences on the Theory and Practice of Software (ETAPS 2000)*, Lecture Notes in Computer Science, pages 319–333, London, UK, March 2000. Springer-Verlag.

[16] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Is code still moving around? looking back at a decade of code mobility. In *29th International Conference on Software Engineering (ICSE COMPANION'07)*, pages 9–20, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Cidiane Lobato, Alessandro Garcia, Alexander Romanovsky, and Carlos de Lucena. An aspect-oriented software architecture for code mobility. *Software: Practice and Experience*, 38(13):1365–1392, 2008.

[18] A. Milanés, N. Rodriguez, and B. Schulze. State of the art in heterogeneous strong migration of computations. *Concurrency and Computation: Practice and Experience*, 20(13):1485–1508, 2008.

[19] Fred Douglis. Ideas ahead of their time. *IEEE Internet Computing*, 12(5):4–6, 2008.

[20] Cristian Mateos, Alejandro Zunino, and Marcelo Campo. Extending MoviLog for supporting Web Services. *Computer Languages, Systems & Structures*, 33(1):11–31, April 2007.

[21] Alessandro Garcia, Carlos de Lucena, and Donald Cowan. Agents in object-oriented software engineering. *Software: Practice and Experience*, 34(5):489–521, 2004.

[22] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[23] Alberto Silva, Artur Romao, Dwight Deugo, and Miguel Mira da Silva. Towards a reference model for surveying mobile agent systems. *Autonomous Agents and Multi-Agent Systems*, 4(3):187–231, September 2001.

[24] Alejandro Zunino, Marcelo Campo, and Cristian Mateos. MoviLog: A platform for Prolog-based strong mobile agents on the WWW. *Revista Iberoamericana de Inteligencia Artificial*, 4(21):83–92, 2003.

[25] Hyacinth Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, September 1996.

[26] Cristian Mateos, Alejandro Zunino, and Marcelo Campo. JGRIM: An approach for easy gridification of applications. *Future Generation Computer Systems*, 24(2):99–118, February 2008.

[27] Pablo Gotthelf, Alejandro Zunino, Cristian Mateos, and Marcelo Campo. GMAC: An overlay multicast network for mobile agent platforms. *Journal of Parallel and Distributed Computing*, 68(8):1081–1096, 2008.

[28] Paul Tarau. Agent oriented logic programming in Jinni 2004. In *ACM Symposium on Applied Computing (SAC' 05)*, pages 1427–1428, New York, NY, USA, 2005. ACM Press.

[29] David Wong, Noemi Paciorek, Tom Walsh, Joe DiCelie, Mike Young, and Bill Peet. Concordia: An infrastructure for collaborating mobile agents. In *1st International Workshop on Mobile Agents (MA'97)*, pages 86–97, 1997.

[30] Danny Lange and Mitsuru Oshima. Mobile agents with Java: The Aglet API. *World Wide Web*, 1(3):111–121, 1998.

[31] Earl Barr, Raju Pandey, and Michael Haungs. MAGE: A distributed programming model. In *21st International Conference on Distributed Computing Systems (ICDCS '01)*, page 303, Washington, DC, USA, 2001. IEEE Computer Society.

[32] A. Chan and Siu-Nam Chuang. MobiPADS: a reflective middleware for context-aware mobile computing. *IEEE Transactions on Software Engineering*, 29(12):1072–1085, 2003.

[33] Rebecca Montanari, Emil Lupu, and Cesare Stefanelli. Policy-based dynamic reconfiguration of mobile-code applications. *Computer*, 37(7):73–80, 2004.

[34] Niranjan Suri, Jeffrey Bradshaw, Maggie Breedy, Paul Groth, Gregory Hill, Renia Jeffers, and Timothy Mitrovich. An overview of the NOMADS mobile agent system. In *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages*, June 2000.

[35] Raffaele Quitadamo, Giacomo Cabri, and Letizia Leonardi. Mobile JikesRVM: A framework to support transparent Java thread migration. *Science of Computer Programming*, 70(2-3):221–240, 2008.

[36] Lorenzo Bettini and Rocco De Nicola. Mobile distributed programming in X-Klaim. In *Formal Methods for Mobile Computing*, volume 3465, pages 29–68. Springer Berlin / Heidelberg, 2005.

[37] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying on the Grid, pages 205–229. Springer, Berlin, Heidelberg, and New York, January 2006.

[38] Alexandru Jugravu and Thomas Fahringer. JavaSymphony, a programming model for the Grid. *Future Generation Computer Systems*, 21(1):239–246, 2005.

[39] Willem van Heiningen, Steve MacDonald, and Tim Brecht. Babylon: Middleware for distributed, parallel, and mobile Java applications. *Concurrency and Computation: Practice and Experience*, 20(10):1195–1224, 2008.

[40] Yu-Cheng Chou, David Ko, and Harry Cheng. An embeddable mobile agent platform supporting runtime code mobility, interaction and coordination of mobile agents and host systems. *Information and Software Technology*, 52(2):185–196, 2010.

[41] Alejandro Zunino and Marcelo Campo. Chronos: A multi-agent system for distributed automatic meeting scheduling. *Expert Systems with Applications*, 36(3):7011–7018, 2009.

[42] J. Cucurull, R. Martí, G. Navarro-Arribas, S. Robles, B. Overeinder, and J. Borrell. Agent mobility architecture based on IEEE-FIPA standards. *Computer Communications*, 32(4):712–729, 2009.