Submission to Internet Computing
Not for Distribution or Attribution: for Review Purposes Only

MARS: a Programmable Coordination Architecture for Mobile Agents

Giacomo Cabri, Letizia Leonardi, Franco Zambonelli (*)
Dipartimento di Scienze dell'Ingegneria – Università di Modena e Reggio Emilia
Via Campi 213/b – 41100 Modena – ITALY
Phone: +39-059-376735 – Fax: +39-059-376799
E-mail: {giacomo.cabri, letizia.leonardi, franco.zambonelli}@unimo.it

(*) Corresponding Author

Abstract

Mobile agents represent a promising technology for the development of Internet applications. However, mobile computational entities introduce peculiar problems w.r.t. the coordination of the application components. The paper outlines the advantages of Linda-like coordination models, and shows how a programmable coordination model based on reactive tuple spaces can provide further desirable features for Internet applications based on mobile agents. Accordingly, the paper presents the design and the implementation of the MARS coordination architecture for Java-based mobile agents. MARS defines Linda-like tuple spaces, which can be programmed to react with specific actions to the accesses made by mobile agents. Several examples in the context of a WWW information retrieval application show the effectiveness of the MARS approach.

Keywords*: Mobile Agents, Coordination, Reactive Tuple Spaces, Java*

1. Introduction

Traditional distributed applications are designed as a set of processes statically assigned to given execution environments and cooperating in a (mostly) network-unaware fashion. The *mobile agent* technology, instead, promotes applications made up of network-aware entities (*agents*) capable of changing their *execution environment* by transferring themselves while executing (*mobility*) [KarT98, FugPV98].

The current interest in mobile agents is broadly justified by the advantages their presence provides in Internet applications: *(i)* mobile agents can dramatically save bandwidth, by moving locally to the resources they need, instead of requiring the transfer of possibly large amounts of data; *(ii)* mobile agents can carry the code to manage remote resources and do not need the remote

availability of a specific server, thus leading to a more flexible application scenario; *(iii)* mobile agents do not require continuous network connection, because interacting entities can move to the same site when the connection is available and then interact without needing further network connection; as a consequence *(iv)* mobile agents intrinsically suit mobile computing systems.

In the last few years, several systems and programming environments have appeared to support the development of distributed applications based on mobile agents [KarT98]. Nevertheless, several issues still need to be faced to make the mobile agent technology widely accepted: secure and efficient execution supports, standardization, appropriate programming languages and coordination models [FugPV98].

In mobile agent applications, one of the fundamental activities is the coordination between one agent and *other agents* or *resources* on the execution environments. However, agent mobility and the openness of the Internet scenario imply problems different from those occurring in traditional distributed systems [CabLZ99]. This paper argues that Linda-like coordination models have the necessary adaptability to suit a wide and heterogeneous networked scenario, and lead to a simple application design. In addition, this paper shows that further flexibility and safety can stem from the definition of a programmable coordination model, in which the effects of interaction events can be programmed to meet specific needs, either of the execution environments or of the application agents.

The above considerations motivate the design and the implementation of the *MARS* (*Mobile Agent Reactive Spaces*) coordination architecture, based on the definition of Linda-like tuple spaces, which agents exploit both to coordinate themselves with other application agents and to access local resources. Moreover, unlike raw Linda-like tuple spaces, MARS tuple spaces can be programmed, either by the local administrator or by the agents themselves, to react with specific actions to the accesses made to them by agents. This characteristic allows specific and stateful coordination policies to be defined and secure agent execution to be enforced, as shown via several examples applied to a case study application.

2. Coordination Models for Mobile Agent Applications

During its nomadic life, an agent needs to coordinate its activities with other entities, whether they are other agents or resources of the execution environments. In particular:

- an application may be made up of several mobile agents, which cooperatively perform a task and are, therefore, in need of exchanging data and knowledge;
- a mobile agent usually roams across remote Internet sites to access resources and services allocated there.

While coordination models have been extensively studied in the context of traditional distributed environments [Adl95, GelC92], mobility and the openness of the Internet environment introduce peculiar problems and needs.

2.1 Coupled versus Uncoupled Coordination Models

Most of Java-agent systems – like *Aglets* [LanO98] and *D'Agents*, formerly Agent-TCL [Kot97] – exploit the client-server coordination model typical of the object-oriented paradigm (even between remote objects via Java *RMI*), and/or specific message-passing APIs. These models, by putting the involved entities in *direct* connection, imply both *spatial coupling* and *temporal coupling* [GelC92, CabLZ99]: agents have both to explicitly name their communication partners (spatial coupling), and to synchronize their activities in some way (temporal coupling). This leads to several drawbacks in mobile agent applications. In fact, if two mobile agents want to communicate directly on a wide scale, they need to be localized by means of complex and highly informed tracing protocols. In addition, repeated interactions between agents require stable network connections; this makes communication highly dependent on network reliability and possibly leads to failure or unpredictable delays. Finally, several mobile agent applications are intrinsically dynamic, because of dynamic agent creation, making it difficult to adopt a spatially coupled model that requires the identification of the communication partners. Thus, in the presence of agents moving in the Internet, direct coordination can be effectively exploited only to rule the accesses to the local resources of an execution environment, via interactions with a local server.

The *meeting-oriented* coordination model, introduced by *Telescript* [Whi97] and also implemented by the *Ara* system [Pei97], solves the spatial coupling problem by making agents interact in the context of abstract meeting points without explicitly naming the partners involved. An agent can open a meeting point that other agents can join (either explicitly or implicitly) and afterwards communicate there and synchronize with each other. Meetings are usually locally constrained to avoid the problems related to non-local communication, i.e., unpredictable delay and unreliability: a meeting takes place in a given execution environment and only local agents can participate in it. Always-open meetings can abstract the role of local services in an execution environment. The major drawback of the meeting model is that it still usually enforces a strict temporal coupling among interacting agents, which are bound to be at the same place at the same time. This mines the intrinsic properties of autonomy and dynamicity of the agents (whose schedule and position cannot be easily predicted) or, if those properties are preserved, makes the risk of missing interactions very high.

Some proposals exploit *blackboard-based* coordination models, making interactions occur via shared data spaces (blackboards), local to each execution environment. These include the *Ambit* model for mobile computations [CarG98] and the *ffMAIN* mobile agent system [DomLD97]. Blackboards can be used as common repositories to store and retrieve messages, as well as a medium to access locally published data and services. Since agents can communicate by leaving messages on blackboards without needing to know where the corresponding receivers are and when they will read the messages, a blackboard enforces temporal uncoupling and suits a mobile scenario in which the position and the schedule of the agents cannot be easily traced. In addition, blackboard models provide for more security than the ones previously described, because any execution environment can fully monitor all the interactions that occur through its local blackboard. However,

as far as agents must agree on a common message identifier to communicate and exchange data via a blackboard (i.e., a file name in Ambit and a URL in ffMAIN), interactions are not spatially uncoupled.

Linda-like coordination models [AhuCG86], exploited in several recent proposals in the context of interactive Internet applications, such as *PageSpace* [Cia98] and *JavaSpaces*, extend blackboard models by introducing associative mechanisms into the shared data space: information is organized in tuples and retrieved in an associative way via a pattern-matching mechanism. Therefore, associative blackboards (e.g., tuple spaces) inherit all the advantages of the blackboards, in terms of security and temporal uncoupling, and go further, by enforcing spatial uncoupling: tuples are accessed by content rather than by identifier, thus requiring no mutual knowledge to let agents coordinate. This well suits mobile agent applications: in a wide and dynamic environment, such as the Internet, a complete and updated knowledge of execution environments and of other application agents may be difficult or even impossible to acquire. As agents would somehow require pattern-matching mechanisms to deal with uncertainty, dynamicity and heterogeneity, it is worthwhile integrating these mechanisms directly in the coordination model, to simplify agent programming and to reduce application complexity.

2.2 Towards Programmable Tuple Spaces

Despite the potential benefits in terms of spatial and temporal uncoupling, the basic Linda coordination model lacks flexibility and control in interactions. In fact, both agent-to-agent and agent-to-execution environment interactions are bound to the built-in – data-oriented – pattern-matching mechanism provided by the tuple spaces, which may not be the most suitable one for all kinds of interactions. On the one hand, complex interaction protocols may be difficult to realize and control by exploiting only Linda pattern-matching. On the other hand, the data-oriented interaction model of Linda does not provide for simple solution to access, besides data, the services provided by a site.

The above problems can be solved thanks to the introduction of programmable tuple space models [OmiZ98, CabLZ99]. In this context, programmability stems from the capability of defining specific computational activities to be triggered in reaction to specific access events to one tuple space. In other words, it implies the capability of deciding which types of access events should trigger which activities into one tuple space, other than the built-in and stateless associative Linda pattern-matching mechanism.

Programmable reactivity of tuple spaces can provide several advantages in mobile agent applications. A site administrator can program reactions to monitor the access events to the local resources and to implement specific security policies for the execution environment, in order to achieve better control and to defend the integrity of the environment from malicious agents. In addition, reactions can be used to implement a dynamic tuple space model, in which tuples are not statically stored data structures but are, instead, dynamically produced on demand. From this point of view, reactions enable a simple data-oriented mechanism for accessing services on a site: the

attempt to read a specific tuple by an agent can trigger the execution of a local service that produces the required tuple as a result. Furthermore, reactions can be used to adapt the effects of the interactions to the specific characteristics of the execution environment and – even more generally – to embed intelligence into the tuple space [OmiZ98].

Further advantages could be provided by giving applications the capability of defining their own coordination rules: agents can carry along the code of the reactions implementing application-specific coordination policies, and install them in the tuple spaces of the sites visited. Of course, this capability would require facing peculiar security issues. Nevertheless, the possibility for an application to define its own coordination rules in the form of reactions in the tuple spaces achieves a sharp separation of concerns between algorithmic and coordination issues [GelC92]. The agents are in charge of embodying the algorithms to solve the problems; the reactions represent the application-specific coordination rules. This can both reduce the agent complexity and simplify the global application design.

A possible drawback of programmable tuple spaces is that irrational programming can lead to a distortion of the application interactions, and can make users lose control over the execution of their application agents. However, on the one hand, the administrator of one site has no interest in altering the application semantics but, instead, (s)he usually has some interest in exploiting the programmability of her/his administered tuple spaces to enrich the availability of data and resources while better protecting them. On the other hand, an application-specific tuple space programming should (be constrained to) confine its effects to the application itself, without influencing agents belonging to other applications.

3. The MARS Coordination Architecture

MARS (*M*obile *A*gent *R*eactive *S*paces), developed at the University of Modena, starts from the above considerations and implements a portable and programmable Linda-like coordination architecture for Java-based mobile agents.

The MARS architecture does not implement a whole new Java agent system. Instead, it has been designed to complement the functionality of already available agent systems, and it is not bound to any specific implementation: it can be associated to different Java-based mobile agent systems with only a slight extension. The current implementation, available for downloading, has already been tested with *Aglets*, *Java-to-go* and *SOMA*.

Globally, the MARS architecture is made up of a multiplicity of independent tuple spaces, each one associated to a node and accessed by the agents locally executing in that node (see figure 1). Therefore, the only requirement to integrate MARS with a mobile agent system is to make the *agent server* – in charge of accepting and executing incoming agents on a node (step *a* of figure 1) – supply agents with a reference to the local MARS tuple space (step *b* of figure 1). Agents on a node can then access the local tuple space via a well-defined set of Linda-like primitives (step *c* in figure 1), while each MARS tuple space can react to these accesses with differentiated behaviors, programmed in a meta-level tuple space to meet specific needs (step *d* of figure 1).

The MARS tuple space can be exploited both for inter-agent communication and to let agents associatively retrieve primitive data items and references to execution environment resources. The local tuple space is the only resource that an agent can directly access onto a node, so the problem of dynamically binding local references to mobile entities [FugPV98] is very limited in MARS.

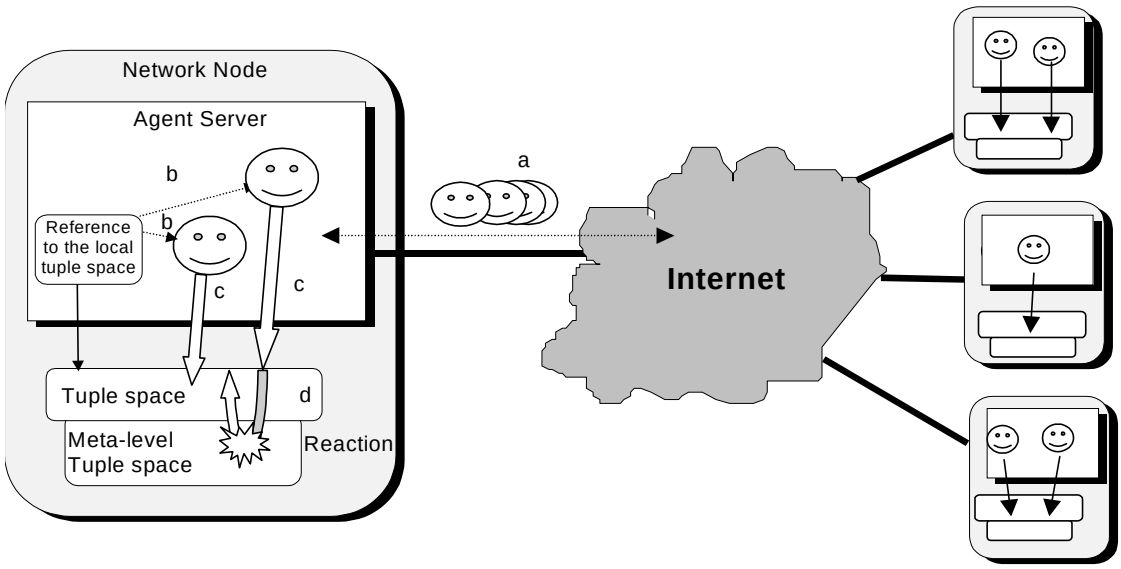

Figure 1. The MARS Architecture (smiles represent mobile agents)

3.1 The MARS Interface

MARS has been designed in compliance with the *JavaSpaces* specification, which is a good candidate to become the *de facto* standard tuple space interface for Java. Therefore, as in JavaSpaces, MARS tuples are Java objects whose instance variables represent the tuple fields. Tuple classes must implement the Entry interface, required for tuple management. The easiest way to do this is to derive a tuple class from the AbstractEntry class (that defines the basic tuple properties by implementing the Entry interface) and to define, as instance variables, the specific tuple fields. Each field of the tuple is a reference to an object that can also represent primitive data type (*wrapper* objects in the Java terminology). Any field of a tuple can have either a defined (actual) or a null (formal) value.

Each MARS tuple space is realized as a Java object, an instance of the Space class, which implements the MARS interface, an extension of the JavaSpace interface (figure 2). To access the tuple space, the following operations are provided:

- write, to put a tuple, supplied as the first parameter, in the space;
- read, to retrieve a tuple from the space, on the basis of a template tuple supplied as the first parameter and to be used as a pattern for the matching mechanism;
- take, which works as the read operation but extracts one matching tuple from the space;
- readAll, to retrieve all matching tuples from the space;
- takeAll, to extract all matching tuples from the space.

For all the operations, the txn parameter can specify a transaction the operation belongs to. The timeout parameter of read, take, readAll and takeAll, specifies the time to wait before the operation returns a null value if no matching tuple is found: while NO_WAIT means to return immediately, 0 means to wait indefinitely. The lease parameter of the write operation sets the lifetime of the written tuple. The readAll and takeAll operations are not present in the JavaSpaces interface and have been added to the MARS interface to avoid agents being forced to perform several reads to retrieve all the needed information, with the risk of retrieving the same tuples several times. This is a well-known problem of tuple space model, due to the non-determinism in the selection of a tuple among multiple matching ones.

```
public interface MARS extends JavaSpace
{
// method interface inherited from JavaSpace
// Lease write(Entry e, Transaction txn, long lease);        // put a tuple into the space
// Entry read(Entry tmpl, Transaction txn, long timeout); // read a matching tuple from the space
// Entry take(Entry tmpl, Transaction txn, long timeout); // extract a matching tuple from the space

// methods added by MARS and not present in the JavaSpace interface
Vector  readAll(Entry tmpl, Transaction txn, long timeout);    // read all matching tuples
Vector takeAll(Entry tmpl, Transaction txn, long timeout);     // extract all matching tuples
}
```

Figure 2. The MARS interface

The template tuple supplied to the read, readAll, take and takeAll operations can have both defined and null values. A tuple *T* matches a template tuple *U* if the defined values of *U* are equal to the corresponding ones in *T*. More in particular, since tuples are objects in MARS (as in JavaSpaces), and since the elements of a tuple can be non-primitive objects, the matching rules must take into account the presence of classes and objects. Then, a template tuple *U* and a tuple *T* match if and only if:

- *U* is an instance of either the class of *T* or of one of its superclasses; this extends the Linda model by permitting a match also between two tuples of different types, provided they belong to the same class hierarchy;
- the defined fields of *U* that represent primitive types (integer, character, boolean, etc.) have the same value of the corresponding fields in *T*;
- the defined non-primitive fields (i.e., objects) of *U* are equal – in their serialized form – to the corresponding ones of *T* (we can mention that two Java objects assume the same serialized form only if each of their instance variables have equal values, recursively including enclosed objects);
- a null value in *T* corresponds to a null value in *U*.

Once a tuple is obtained from the space, its actual field objects can be accessed as any other Java object.

As an example, let us suppose that a file of the local file system is represented by a tuple of this type (String PathName, String Extension, Date ModificationTime, File ActualFile). Then, a template tuple (null, “html”, null, null) matches with the tuple (“/usr/local/htdocs/index”, “html”, 02/01/98:17.34.20, File@[012a34]) and also with (“/home/java-to-go-server/mobile_agents”, “html”, 01/10/97:14.12.54,

File@[0b6c52]); a template tuple ("/home/java-to-go-server/mobile_agents", "html", null, null) matches only with the latter tuple.

3.2 The Reactive Model

Linda, in its first definition, provided for a limited form of reactivity, via a specific operation (called *eval*) used for the dynamic evaluation of tuples [AhuCG86]. However, due to both the unclear semantics of the eval and the lack of a suitable model for controlling the computations performed in a tuple space, most implementations (including JavaSpaces) rules out the eval operation, as well as any form of reactivity. MARS, while discarding the eval implementation, recognizes the need for tuple space reactivity and defines a flexible and controllable architecture to associate programmable *reactions* to the events on the tuple space, i.e., to the tuple space accesses performed by agents.

MARS reactions are stateful objects with a known method (namely, the reaction method), which can access the tuple space itself, change its content, and influence the effects of the operations performed by agents.

The association of reactions to access events is represented via *4-ples* stored in a "*meta-level*" tuple space. A meta-level tuple has the form of *(Rct, T, Op, I)*: it means that the reaction method of the object *Rct* will be triggered when an agent with identity *I* invokes the operation *Op* on a tuple matching *T*. Putting and extracting tuples from the meta-level tuple space means installing and de-installing, respectively, reactions associated to events at the base-level tuple space.

A meta-level 4-ple can have some non-defined values, in which case it associates the specified reaction to all the access events that match it. For example, the 4-ple (ReactionObj, null, read, null) in the meta-level tuple space associates the reaction of the ReactionObj instance to all read operations, whatever the tuple type and content and the agent identity.

The meta-level tuple space follows associative mechanisms similar to the ones of the "base-level" tuple space. A meta-level pattern-matching mechanism is activated for any access to the base-level tuple space, to detect the presence of reactions to be executed (i.e., of meta-level tuples matching the access event) and trigger their execution. For example, when an agent with identity *I* invokes a single-tuple input operation *Op* (i.e., read or take) supplying a template tuple *U*, MARS executes the following steps (see figure 3):

1. it issues the pattern matching mechanism of the base-level tuple space to identify a tuple *T* that matches with *U;*
2. it executes a readAll in the meta-level tuple space by providing the 4-ple (null, T, Op, I), where T is the tuple *T* that has matched in step 1, if any, the template *U* otherwise;

3a. if a matching 4-ple is found in the meta-level, MARS triggers the execution of the corresponding reaction. The reaction method receives the tuple T, as well as the Op and I values, as parameters, and it is expected to return a tuple as a result.

3b. if no matching meta-level 4-ples are found, MARS lets the invoked operation *Op* proceed according to its normal semantics.

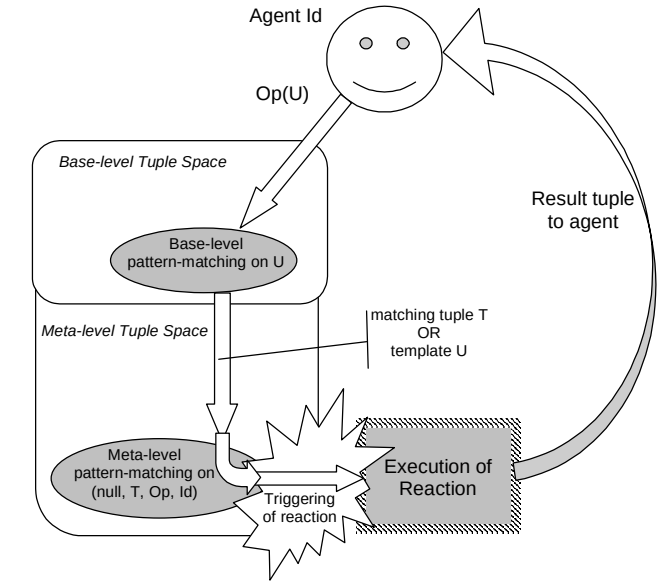

Figure 3. The MARS meta-level activities

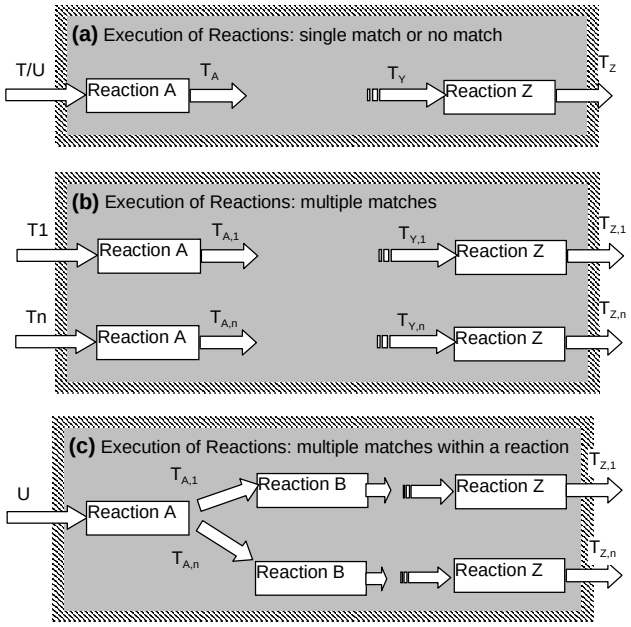

Figure 4. The cases of (a) multiple reactions; (b) multiple reactions and multiple matching tuples; (c) multiple reactions and multiple matches within a reaction

If several 4-ples satisfy the meta-level matching mechanism in step 2, all the corresponding reactions are executed in a pipeline (see figure 4a), accordingly to their installation order (since the order in which the reactions are executed in the pipeline may influence the final result, MARS rules out non-determinism for meta-level matches).

In the case of a readAll or a takeAll invoked in the base-level tuple space, multiple matches can be produced either in step 1 or by one of the reactions triggered in step 3a. If multiple matches are produced in step 1, a pipeline of matching reactions is associated to each matching tuple (see figure 4b). If no matches are produced in step 1, a single pipeline of reactions is activated; as soon as a reaction in the pipeline produces multiple result tuples, the rest of the pipeline is replicated for

each result tuple (see figure 4c).

In the case of a write operation, MARS starts directly from step 2 and uses the tuple *T*, parameter of the write operation itself, in the meta-level 4-ple (null, T, write, I).

A reaction has access to the base-level tuple space and can perform any kind of operation on it (however, operations on the base-level tuple space performed within a reaction do not issue any reaction, to avoid endless recursion). This can be used to exploit the base-level tuple space as a repository of reaction state information, in addition to the state that the reaction object can store as part of its state. As a consequence, the behavior of the reaction can depend both on the actual content of the tuple space and on past access events. Also, when the reaction executes, it has the availability of either the result of the matching mechanism issued by the associated operation, or the template with which an input operation was invoked. Then, the reaction can influence the effects of operations and, for example, can return to the invoking agent different tuples than the ones resulting from the normal pattern-matching mechanisms.

The above characteristics of the MARS reactive model lead to great flexibility, as the application example section will show, and substantially distinguish it from the *notify* mechanism of JavaSpaces. In fact, far from introducing reactivity into tuple spaces, the notify mechanism can simply be used to signal external objects about the insertion of new tuples into a tuple space. Thus, it does not allow other access events to be monitored but writes and does not give the possibility of controlling and influencing the effects of tuple space operations, since the activities possibly triggered in a notified object are outside the space and independent of its internal activity.

3.3 The Security Model

Security is a very important issue in the context of mobile agent applications. In MARS, security concerns protecting the tuple spaces from unauthorized or malicious accesses. MARS assumes that the mobile agent system – like most of the available systems – provides for agent identification and authentication before allowing any agent to execute on one node. On this basis, the main security mechanism provided by MARS is the association of one *Access Control List* (ACL) to each tuple. Such ACL defines *who* can do *what* on the corresponding tuple, and the system administrator can decide whether a given operation on a given tuple is authorized or not.

Starting from the ACL mechanism, MARS integrates security policy based on *roles*. Agent identities can be mapped into roles, in which case MARS uses roles as *de facto* identities of agents. Translating identities into a few well-defined roles simplifies the management of the ACLs and leads to a better uncoupling between the agent system and MARS. As an example, three roles of a very general nature, which can be defined for any tuple space, are *reader*, *writer* and *manager*. Agents with a reader role can read tuples of the space, but they can neither insert new tuples nor extract already stored tuples. Agents with a writer role have the right to read tuples and store their own tuples in the space, but they cannot extract tuples stored by agents with a different identity. Agents with a manager role, typically owned by the local administrator, have full rights on the space: they can read, write and extract any tuple. In addition, they also have full access to the meta-

level tuple space, thus being enabled to dynamically install and uninstall reactions.

Other roles can be defined which give or deny access to specific classes of tuple and which allow agents to install and uninstall reactions only with regard to a limited set of access events. In general, the administrator of one site should define proper role authorization policies for external application agents, so as to guarantee that an external agent cannot influence – by installing/de-installing reactions – the activities of other agents but those of its specific application.

4. Using MARS in Mobile Agent Applications

An application example in the area of WWW information retrieval can show the ease of use of MARS and its capability of leading to a simpler and more flexible application design than other coordination models. The application exploits mobile agents to reach remote Internet sites and locally access HTML pages of interest, analyzing them and extracting the required information without any need to transfer the pages over the network. For instance, a searcher agent sent to a remote site can analyze the local WWW pages and come back with the URLs of the pages that contain a specific keyword. To speed up the research, the application can be shaped after a tree of concurrent searcher agents. If a searcher agent on a site finds HTML links to other possibly interesting pages on different sites, it clones itself and has the clones follow these links, to recursively continue the search work on different sites (figure 5). In this context, agent-to-execution environment coordination is needed to make agents access and retrieve information on a site, while inter-agent coordination is needed to avoid duplicated work performed by different clones on the same sites, as could occur in the common case of cross-references in HTML pages.

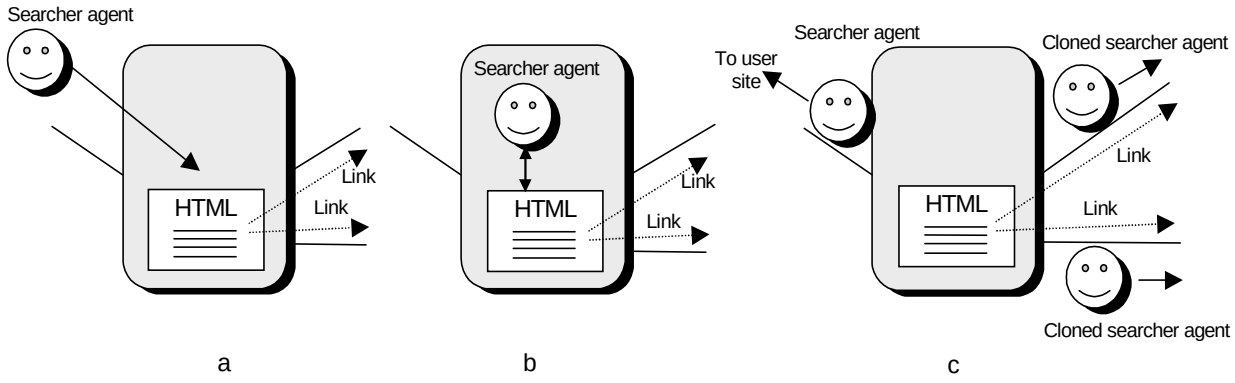

Figure 5. An agent arrives at a site (a), accesses the local HTML pages (b) and creates a new agent for any link to be followed before returning to the user site (c).

4.1 Agent-to-Execution Environment Coordination

When relying on direct or meeting-oriented coordination, the access to the local resources of an execution environment is to be based on local servers, which are requested to provide appropriate services to retrieve the data of interest, the HTML pages in the specific example. However, a server may not always provide the appropriate services needed for an application. For example, most of

today's WWW servers cannot (or are not configured to) provide the index of the URLs available on a site. So, an agent is forced to explicitly navigate page by page in the site, from the root HTML page to the leaves, to retrieve all available HTML pages. Code mobility can make it possible for agents to carry the code of the needed services along and install them in the sites visited. This capability, however, introduces security issues and requires exception handling, in case the execution of the services on a site fails. As a further drawback, the solution forces agents to adopt a control-oriented point of view, i.e., to request services while they require data/files. With the adoption of a blackboard or a tuple space on each site, HTML pages on a site can be accessed without requiring the presence of peculiar services and in a more natural data-oriented style.

```
class FileEntry extends AbstractEntry      // AbstractEntry is the root class of the tuple hierarchy
{   // tuple fields
    public String PathName;                // pathname of the file represented by this tuple
    public String Extension;               // representative of the file type
    public Date ModificationTime;          // date of the last update of the file
    public File ActualFile;                // a file object to access the content of the file

    // constructor of the tuple. The order of the parameters defines the tuple field order
    public FileEntry(String name, String extension, Date modificationTime, File file)
        {   PathName = name; Extension = extension;
            ModificationTime = modificationTime; ActualFile = file;
        }
}
```

Figure 6. The FileEntry class for the tuples representing files in the tuple space

In the case of MARS, the local execution environments can provide access to the HTML pages via tuples that make available general file information, such as pathname, extension and modification time, and that contain a reference to a File object to be used to access the content of the file. Instances of the FileEntry class shown in figure 6 can represent this tuples.

```
...
FileEntry FilePattern = new FileEntry(null, "html", null, null);  // creation of the template tuple

Vector HTMLFiles = LocalSpace.readAll(FilePattern, null, NO_WAIT);
// read all matching tuples and return a vector of tuples

if (HTMLFiles.isEmpty())  // no matching tuple is found
    terminate();       // the agent has nothing to return to the user and can terminate
else
    for (int i = 0; i < HTMLFiles.size(); i++)      // for each matching tuple
    {   FileEntry Hfile = (FileEntry)HTMLFiles.elementAt(i);   // Hfile: tuple representing the file
        if (this.SearchKeyword(keyword, Hfile.ActualFile))     // search for the keyword in the file
                {   FoundFiles.addElements(LocalHost, Hfile); // store the file in a private vector
                                                               // to be returned to the user  and
                    this.SearchLink_and_Clone(Hfile.ActualFile);   // search for remote links and
                }                                                  // send clones to remote sites
    }
    go_to(home);        //return to the user site
...
```

Figure 7. Code of the searcher agents (fragment)

Agents that look for HTML pages can obtain references to them by accessing the tuple spaces,

in a suitable data-oriented way. A fragment of the Java code for the agents of the application example is shown in figure 7. The searcher agents invoke the readAll operation in a non-blocking mode, by providing a template FileEntry tuple in which only the Extension field is defined and has the "html" value. The operation returns all matching tuples – i.e., the ones representing HTML documents – in a vector. For each matching tuple, the agent accesses the corresponding file via the ActualFile field, to search for the keywords of interest in its content. Finally, the agent searches for remote links in the files and clones itself to follow these links before going back home.

Note that, in the case of non-associative blackboards, there would not have been a way for an agent to selectively access HTML files only: instead, an agent would have been forced to read from the blackboard all the data, and select HTML files by programming a pattern-matching algorithm on the file name extension.

The above basic scheme for agent-to-execution environment coordination can be made more flexible and secure by exploiting the MARS reactivity, without influencing the agent code.

As a first example, let us suppose one site stores all its HTML pages in files having "htm" extension instead of "html". In this case, the administrator can exploit reactivity to enable the access to its HTML files to agents requesting the more standard "html" extension, without being forced to change local filenames or duplicate the FileEntry tuples in both "html" and "htm" forms, and without forcing agents to explicitly deal with this kind of heterogeneity. To this end, (s)he can install a reaction that transforms any request for file tuples with "html" extension into a request for tuples having "htm" as extension field. This reaction, implemented by the HTML2HTM shown in figure 8, can be associated to the specific requests for "html" tuples via the meta-level tuple: (HTML2HTM_instance, (null, "html", null, null), readAll, null).

```
class HTML2HTM implements Reactivity
{
public Entry[ ] reaction(Space s, Entry Fe, Operation Op, Identity Id)
// no match already occurred if the site has only htm files
{ Fe.Extension = "htm";  // modifies the extension of the required files
  return s.readAll(Fe, null, NO_WAIT);
}}
```

Figure 8. The HTML2HTM reaction class

As a second example, let us suppose that either a badly programmed or an intentionally malicious agent (which of the two categories an agent actually fits into cannot be distinguished from the hosting environment's perspective) tries to perform a takeAll operation on the "html" tuples. In this case, the system administrator can decide not to raise any exception, and to let the agent read the content of the matching tuples without deleting them. To this purpose, (s)he can install a reaction that transparently transforms any disruptive takeAll operation, performed by a foreign agent on the FileEntry tuples, into a non-disruptive readAll operation. This reaction, implemented by the TransformTake class in figure 9, which also includes the recording of the bad attempt on a system register, can be installed by writing the 4-ple (TransformTake_instance, (null, "html", null, null), takeAll, null) in the meta-level tuple space.

The administrator can combine the HTML2HTM and TransformTake reactions by inserting the 4-ple (HTML2HTM_instance, (null, “html”, null, null), takeAll, null) into the tuple space, before the above introduced 4-ple for the TransformTake reaction. This makes the tuple space reacts with an HTML2HTM-TransformTake pipeline to takeAll operations requesting for “html” tuples, and has the TransformTake reaction executed for each of the “htm” tuples returned by the HTML2HTM reaction (as in figure 4c).

We emphasize that JavaSpaces cannot achieve the behavior of the above reactions in any way, since the notify mechanism can neither capture input events nor modify their effects.

```
class TransformTake implements Reactivity
{
public Entry reaction(Space s, Entry Fe, Operation Op, Identity Id)
// the parameters represent, in order:
//          the reference to the local tuple space
//          the reference to the matching tuple
//          the operation type
//          the identity (or the assigned role, if any) of the agent performing the operation

{ if(matched(Fe)) { // if a match has been produced
    if (Id.equals(manager))          // check for the identify of the agent performing the operation
          return s.take(Fe, null, NO_WAIT);// the tuple is deleted from the space
                                           // because the administrator has full rights
    else  {SecurityRegister.add(“takeAll”, Fe, Id);   // log the access
          return Fe }       // the tuple is returned but it not extracted
  else return null; }         // no match has been produced
}}
```

Figure 9. The TransformTake reaction class

4.2 Inter-Agent Coordination

In the case of inter-agent coordination – needed to avoid duplicated work on a site – the adoption of a direct coordination model forces odd design solutions for the case study. Because agents are dynamically created and are autonomous in their movements, it is not possible for an agent to know who and where other application agents are. Then, the only way for an agent to know whether other agents that have already visited a site exist is to integrate a specialized directory server in the application design. The directory server, allocated for example in the user home site, keeps track of sites already visited and has to be remotely accessed by application agents to know whether a site has been visited or not and to update the list of sites visited. This solution not only complicates the application design but, by requiring non-local communications, also makes it less efficient and reliable.

In the case of meeting-oriented coordination, a more distributed solution can be sketched. However, this requires the introduction of further special-purpose entities in the application. In particular, when an agent has explored a site, it has to create another agent, i.e., a "meeting" agent, which is forced to reside on the creation site to notify – via an *ad hoc* opened meeting-point– further incoming agents about a previous visit. Since the meeting agents open meeting-points to basically act as shared information spaces, if the coordination model itself is based on shared dataspaces – as

in the case of blackboards or tuple spaces – inter-agent coordination can be simply realized without peculiar design solutions.

In particular, in the case of MARS, any searcher agent arriving on a site can check in the local tuple space for a "marker" tuple, written by another agent of the same application and having the form (my_application_id, "visited"). If the agent obtains the tuple, it knows that the site has already been visited and terminates; otherwise, it is in charge of putting the marker tuple in the tuple space.

To realize the above scheme, searcher agents must be assigned a role that permits them to write tuple in the tuple space, i.e., a writer role. However, this makes a node at risk of being overwhelmed with tuples left by agents that no one will ever use and that no one will ever care to delete from the space. To face this problem, the administrator can decide to exploit the MARS reactivity to install a reaction that overrides the write operation performed by the agents and write the tuple in the tuple space by specifying a site-specific allowed lifetime, disregarding the lifetime possibly specified by the writer agent via the lease parameter. This reaction, implemented by the LifeTime class shown in figure 10, can be associated to all write operations performed by the agents with a writer role, by writing the 4-ple (LifeTime_instance, (null, null), write, writer) in the meta-level tuple space.

```
class LifeTime implements Reactivity {
  long LifeTime;

  public LifeTime(long Lifetime)    // constructor
  {         this.Lifetime = Lifetime;}

  public Entry reaction(Space s, Entry Fe, Operation Op, Identity Id) {
            return s.write(Fe, null, Lifetime);
  }
}
```

Figure 10. The LifeTime reaction class

```
class IncrementalVisit implements Reactivity
{
  private Date visit;          // date of the last visit
  public IncrementalVisit() // constructor
  {  visit = new Date(); }     // when the reaction is installed by an agent, the date of last visit is
                               // automatically set by the constructor

  public Entry reaction(Space s, Entry Fe, Operation Op, Identity Id)
  {  if (s.matched(Fe) && !(FileEntry)Fe.ModificationTime.before(visit))
     // is this a matching document having been modified after last visit?
        {  visit = new Date();          // set the time of the last visit
           return Fe;  }                // return the tuple representing the updated page
     else
           return null;                 // if no, no tuple has to be returned
  }
}
```

Figure 11. The IncrementalVisit reaction class

A more sophisticated way of avoiding duplicated work on the same site can be achieved by allowing agents to install application-specific reactions associated to access events performed by

agents of the same application and related to “html” tuples. In this case, application agents could install a stateful reaction that does not simply avoid the retrieval of duplicated information, but also takes into account the possible update of HTML pages. In particular, when an agent accesses the tuple space to retrieve the references to the HTML pages, the reaction can check – for any matching FileEntry tuple – whether the corresponding file has been modified or not since the last visit of another agent of the same application. If the file has not been modified, the corresponding tuple is not returned to the agent. This reaction, implemented by the IncrementalVisit reaction class shown in figure 11, can be installed by writing the 4-ple (IncrementalVisit_instance, (null, “html”, null, null), readAll, my_application_id) in the meta-level tuple space.

5. Performance Evaluation

We have performed several measurements to evaluate the times required to access one MARS tuple space and, in particular, to evaluate the overhead introduced by the MARS reactive model. Figure 12 shows the time needed by an Aglets agent, running on a Sun ULTRA 10, to perform a read operation in the local MARS tuple space and return a matching tuple, in different cases:

- by completely deactivating the meta-level activities (*read – passive tuple space* case), as happens in any non-reactive tuple space implementation;
- by activating the meta-level activities, without any reaction installed (*read – 0 reaction issued – 0 reactions installed* case);
- by activating the meta-level activities, with different numbers of reactions installed, and by making the read operation issue one null-body reaction (*read – 1 reaction issued – 1, 20, 50 reactions installed* cases).

For all cases, as expected, the access times to the MARS tuple space increase nearly logarithmically with the number of tuples stored in the base-level tuple space (from 4 to 10 ms in the tests performed), due to the larger amount of information to be dealt with in the pattern-matching process.

By comparing the access times to the tuple space with and without the meta-level matching mechanisms activated, one can see that the overhead introduced by the MARS reactive model is very limited, independently of the global number of tuples in the base-level. The overhead introduced by the activities of the meta-level tuple space is about 10% when there are no reactions (i.e., meta-level tuples) installed, and grows very slightly with the increase of the number of reactions installed. For instance, in the case that 50 reactions are installed in the tuple space (a number that, in our opinion, overestimates the normal situation of a tuple space), and a null-body one is triggered by the read, the overhead introduced is well below 30%. The above measurements of the overhead introduced by reactivity on the specific MARS implementation are likely to apply to other tuple space implementations (for example JavaSpaces), whenever they are extended towards programmability.

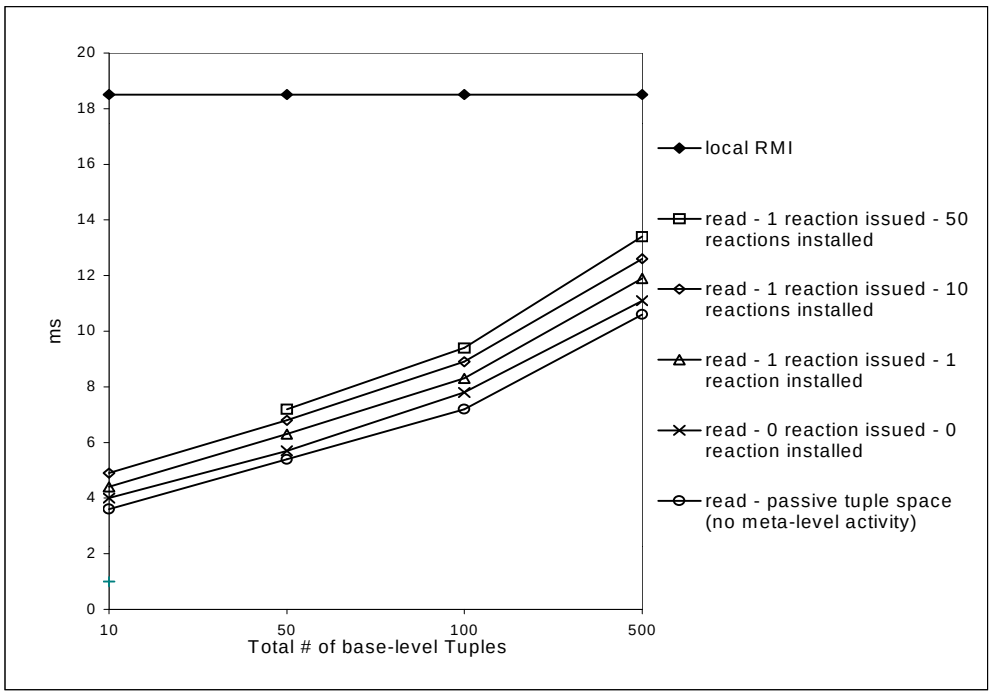

Figure 12. Overhead introduced by reactions (the case of 50 reactions starts from 50 tuples in the base-level space)

By considering other interaction mechanisms that can be possibly exploited by Java mobile agents, the advantages of MARS-mediated interactions become clearer. For instance, a null-body local RMI costs, on the same node, at least 18 ms, well over the average time for a read in MARS. In the case of remote interactions, an RMI between a SUN Ultra 10 and a SPARCstation 5 connected through a 10Mb Ethernet costs about 30 ms; even worse, the proxy-mediated Aglets messaging system makes this costs increase to 37 ms. By considering that the round-trip time for the migration of an Aglets agent of 2.5 Kb is, in the same architecture, about 250 ms, it is easy to calculate that, as soon as an agent needs more than a dozen interactions with the resources of a site to carry out its task, it is cheaper to migrate the agent to that site and let it interact locally via MARS, rather than letting it interact remotely.

6. Related Work

Apart from JavaSpaces, whose main features have been already discussed thoroughout the paper, other proposals exist which exploit tuple spaces in the context of agent-based Internet applications.

The *PageSpace* architecture for interactive Web applications exploits Linda-like coordination [Cia98]: tuple spaces can be used by both mobile and fixed agents to store and associatively retrieve object references; furthermore, agents can create private tuple spaces to interact privately without affecting hosting execution environments. Though not reactive in itself, PageSpace identifies the limit of the raw Linda model and defines special purpose agents, in charge of accessing the space and changing its content, to influence the coordination activities of application agents. However, the

potentialities of these special-purpose agents are very limited, in terms of both monitoring interaction events and tuning their effects.

The *T Spaces* project at IBM defines Linda-like interaction spaces to be used as general-purpose information repositories for networked and mobile applications. In particular, rather than aiming at defining agent-oriented coordination media, T Spaces aims at providing a powerful and standard interface for accessing large amounts of information organized as a database. For this reason, T Spaces recognizes the limits of the basic Linda model and integrates a peculiar form of programmability, by enabling new behaviors to be added to a tuple space. However, these new behaviors are added in terms of new admissible operations on a tuple space (typically complex queries), rather than by programming the effects of the basic Linda operations, as in MARS. In our opinion, this makes T Spaces less usable in the open Internet environment, since it requires application agents either to be aware of the operations available in a given tuple space or to somehow dynamically acquire this knowledge.

An important characteristic that distinguishes MARS from all of the above systems concerns the way agents refer to tuple spaces. In MARS, each agent holds a single reference that is implicitly bound to the tuple space of the local execution environment. Instead, in both PageSpace and T Spaces (as well as in JavaSpaces) agents can refer to multiple tuple spaces, whether local or remote, via different tuple space references, and access them in a location-unaware fashion. On the one hand, this can make it more difficult to manage applications, since the agent code has to explicitly manage tuple space references. On the other hand, this makes agent execution less controllable and reliable, since interactions occur transparently to the location of agents and tuple spaces.

The *TuCSoN* model [OmiZ98], developed in the context of an affiliated research project, faces the above problem by making agents refer to tuple spaces via URLs, as an Internet service, thus enforcing network-awareness. With regard to the tuple space model, TuCSoN resembles MARS in its full programming capability of tuple spaces. However, TuCSoN defines programmable logic tuple spaces where both tuples and reactions are expressed in terms of untyped first-order logic terms. This characteristic of TuCSoN complements MARS and makes it very well suited for the development of applications based on intelligent information agents, in charge of accessing and managing large and heterogeneous information sources.

The *MOLE* system [Bau98], although defining a meeting-oriented coordination model rather than a tuple-based one, is worth mentioning in this context. MOLE meetings (called ”sessions”) occur via shared, non-mobile, objects, which agents must access to send/receive messages. However, unlike Telescript and Ara ones, MOLE meetings enforce temporal uncoupling – by permitting asynchronous notification of messages to agents – and can be programmed in order to integrate specific policies for the management of the messages exchanged. These characteristics lead to an uncoupled and programmable coordination model, like MARS. However, MOLE enforces a control-oriented coordination style, in contrast with the data-oriented one of MARS, and requires the application-level definition of meeting-points.

7. Conclusions and Work in Progress

Linda-like coordination models, possibly enriched with the capability of programming the behavior of the tuple spaces, well suit mobile agents applications and lead to simple and flexible application design. In this context, MARS defines a general and portable coordination architecture, which facilitates the design and development of Internet applications based on Java mobile agents.

We are presently extending MARS for integration into a proxy-based framework for computer supported cooperative work. This integration is going to lead to a general framework for the execution of interactive Web applications based on mobile agents [Cia98]. In addition, we are currently trying to solve some open security problems that may arise if foreign application agents are allowed to program tuple spaces. In particular, while MARS can currently confine the effects of agent-installed reactions, it cannot avoid spamming of endless and computationally intensive reactions. Moreover, it is not clear which garbage collection mechanisms can be defined to avoid a site being overwhelmed with reactions installed by application agents that are no longer useful. However, these seem to be general open problems of the mobile agent technology rather than peculiar problems of the MARS reactive model.

Acknowledgments

The authors thank the anonymous referees for their useful suggestions. This work has been supported by the Italian Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the MOSAICO Project "Design Methodologies and Tools of High Performance Systems for Distributed Applications".

References

[Adl95] R. M. Adler, “Distributed Coordination Models for Client-Server Computing”, *IEEE Computer*, Vol. 29, No. 4, pp. 14-22, April 1995.

[AhuCG86] S. Ahuja, N. Carriero, D. Gelernter, “Linda and Friends”, *IEEE Computer*, Vol. 19, No. 8, pp. 26-34, August 1986.

[Bau98] J. Baumann, F. Hohl, K. Rothermel, M. Straßer, “Mole - Concepts of a Mobile Agent System”, *The World Wide Web Journal*, Vol. 1, No. 3, pp. 123-137, 1998.

[CabLZ98] G. Cabri, L. Leonardi, F. Zambonelli, “Coordination Models for Internet Applications based on Mobile Agents”, *IEEE Computer*, 1999, to appear.

[CarG98] L. Cardelli, D. Gordon, “Mobile Ambients”, *Foundations of Software Science and Computational Structures*, Lecture Notes in Computer Science, No. 1378, Springer-Verlag (D), pp. 140-155, 1998.

[Cia98] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, A. Knoche, “Coordinating Multi-Agents Applications on the WWW: a Reference Architecture”, *IEEE Transactions on Software Engineering*, Vol. 24, No. 8, pp. 362-375, May 1998.

[DomLD97] P. Domel, A. Lingnau, O. Drobnik, “Mobile Agent Interaction in Heterogeneous Environment”, *1st International Workshop on Mobile Agents*, Lecture Notes in

Computer Science, Springer-Verlag (D), No. 1219, pp. 136-148, April 1997.

[FugPV98] A. Fuggetta, G. Picco, G. Vigna, "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, Vol. 24, No. 5, pp. 352-361, May 1998.

[GelC92] D. Gelernter, N. Carriero, "Coordination Languages and Their Significance", *Communications of the ACM*, Vol. 35, No. 2, pp. 96-107, February 1992.

[KarT98] N. M. Karnik, A. R. Tripathi, "Design Issues in Mobile-Agent Programming Systems", *IEEE Concurrency*, Vol. 6, No. 3, pp. 52-61, July-September 1998.

[KinZ97] J. Kiniry, D. Zimmermann, "A Hands-on Look at Java Mobile Agents", *IEEE Internet Computing*, Vol. 1, No. 4, pp. 21-33, July - August 1997.

[Kot97] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, G. Cybenko, "Agent TCL: Targeting the Needs of Mobile Computers", *IEEE Internet Computing*, Vol. 1, No. 4, pp. 58-67, July - August 1997.

[LanO98] D. B. Lange, M. Oshima, "Programming and Deploying Java™ Mobile Agents with Aglets™", Addison-Wesley, Reading (MA), August 1998.

[OmiZ98] A. Omicini, F. Zambonelli, "TuCSoN: a Coordination Model for Mobile Agents", Journal of Internet Research, Vol. 8, No. 5, pp. 400-413, 1998.

[Pei97] H. Peine, "Ara - Agents for Remote Action", in W. R. Cockayne and M. Zyda eds.: Mobile Agents: Explanations and Examples, Manning/Prentice Hall, 1997.

[Whi97] J. White, "Mobile Agents", in J. Bradshaw ed.: Software Agents, AAAI Press, Menlo Park (CA), pp. 437-472, 1997.

Selected URLs on Mobile Agents and Coordination

Ambit	http://www.luca.demon.co.uk/Ambit/Ambit.html
ARA	http://www.uni-kl.de/AG-Nehmer/index_e.html
D'Agents (Agent-TCL)	http://www.cs.dartmouth.edu/~agent
General Magic	http://www.genmagic.com
IBM Aglets	http://aglets.trl.ibm.co.jp
JavaSpaces	http://chatsubo.javasoft.com
MARS	http://sirio.dsi.unimo.it/MOON
MOLE	http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html
PageSpace	http://flp.cs.tu-berlin.de/~pagespc/
SOMA	http://www-lia.deis.unibo.it/Software/MA/
T Spaces	http://www.almaden.ibm.com/TSpaces
TuCSoN	http://www-lia.deis.unibo.it/Resarch/TuCSoN