MARS:
A Programmable Coordination Architecture for Mobile Agents

GIACOMO CABRI, LETIZIA LEONARDI, AND FRANCO ZAMBONELLI
University of Modena and Reggio Emilia

Mobile agents offer much promise, but agent mobility and Internet openness make coordination more difficult. Mobile Agent Reactive Spaces, a Linda-like coordination architecture with programming features, can handle a heterogeneous network while still allowing simple and flexible application design.

Traditional distributed applications are designed as sets of processes—mostly network-unaware—cooperating within assigned execution environments. Mobile agent technology, however, promotes the design of applications made up of network-aware entities that can change their execution environment by transferring themselves while executing.[1,2] Current interest in mobile agents stems from the advantages they provide in Internet applications:

- bandwidth savings because they can move computation to the data,
- flexibility because they do not require the remote availability of specific code, and
- suitability and reliability for mobile computing because they do not require continuous network connections.

Several systems and programming environments now support mobile-agent-based distributed application development.[2] For wide acceptance and deployment, however, we need tools that can exploit mobility while providing secure and efficient execution support, standardization, and coordination models.[1]

Coordination between an agent and other agents and resources in the execution environment is a fundamental activity in mobile agent applications. Coordination models have been extensively studied in the context of traditional distributed systems;[3,4] however, agent mobility and the openness of the Internet introduce new problems related to naming and tracing agents roaming in a heterogeneous environment.[5]

We believe that a Linda-like coordination model—relying on shared data paces accessed in an associative way—is adaptable enough for a wide and heterogeneous network scenario and still allows simple application design. Further, allowing interaction events to be programmed for specific agent or execution environment needs allows additional flexibility and security. The Mobile Agent Reactive Spaces (MARS) coordination architecture, developed at the University of Modena and Reggio Emilia, provides both features.

 1089-7801/00/$10.00 ©2000 IEEE

COORDINATION MODELS

Mobile agent coordination models fall into four categories: client-server, meeting oriented, blackboard based, and Linda-like.

Client-Server Model

Most Java-agent systems—such as Aglets[6] and D'Agents (formerly Agent-TCL)[7]—use a *client-server* coordination model, specific message-passing APIs, or both. By directly connecting the involved entities, this model entails both *spatial coupling*, where agents have to explicitly name their communication partners, and *temporal coupling*, where agents must synchronize their activities.[4,5] This leads to several drawbacks for mobile agent applications:

- To communicate, mobile agents must use complex and highly informed tracing protocols.
- Repeated interactions between agents require stable network connections.
- Mobile agent applications often dynamically create new agents, which makes it difficult to identify communication partners in a spatially coupled model.

Thus, client-server coordination is best reserved for sharing information in a local execution environment.

Meeting-Oriented Model

The *meeting-oriented* coordination model, introduced by Telescript[8] and also implemented by the Ara system,[9] solves the spatial coupling problem by letting agents interact at abstract meeting points without explicitly naming involved partners. An agent can open a meeting, which other agents can join, explicitly or implicitly, and communicate and synchronize with each other. Meetings are usually locally constrained to avoid remote communication problems such as unpredictable delay and unreliability.

The meeting-oriented model's major drawback is that it still requires a strict temporal coupling of interacting agents, forcing them to be at the same place at the same time. This either undermines the autonomy and dynamics of agents, whose schedule and position cannot be easily predicted, or if those properties are preserved, increases the risk of missing interactions.

Blackboard-Based Model

Some proposals—including the Ambit model for mobile computations[10] and the ffMAIN mobile agent system[11]—use the *blackboard-based* coordination model, where interactions occur in shared data spaces, or blackboards, local to each execution environment. Blackboards serve as both common repositories for messages and as sources for accessing locally published data and services.

Since agents communicate by leaving messages on blackboards without knowing where receivers are or when they'll read the messages, blackboards allow temporal uncoupling and thus suit mobile computing. Blackboards also provide more security than other models because execution environments can monitor all local blackboard interactions. However, because agents must agree on a common message identifier to communicate and exchange data—for example, a file name in Ambit or a URL in ffMAIN—interactions are not spatially uncoupled.

Coordination among agents and resources is a fundamental activity in mobile agent applications.

Linda-like Model

Several recent proposals for interactive Internet applications—including PageSpace[12] and JavaSpaces[13]—use a *Linda-like* coordination model,[14] which extends the blackboard-based model by introducing associative mechanisms into the shared data space. Information is organized in tuples and retrieved in an associative way through a pattern-matching mechanism.

Therefore, in addition to having all the advantages of blackboards, associative blackboards—tuple spaces—also allow spatial uncoupling. Because tuples are accessed by content rather than by identifier, agent coordination requires little mutual knowledge, which suits mobile agent applications. In fact, in the wide and dynamic environment of the Internet, it is difficult or even impossible for agents to acquire complete and up-to-date knowledge of other agents or execution environments.

PROGRAMMABLE TUPLE SPACES

Despite allowing spatial and temporal uncoupling, the Linda-like coordination model lacks flexibility and interaction control. In fact, both agent-to-

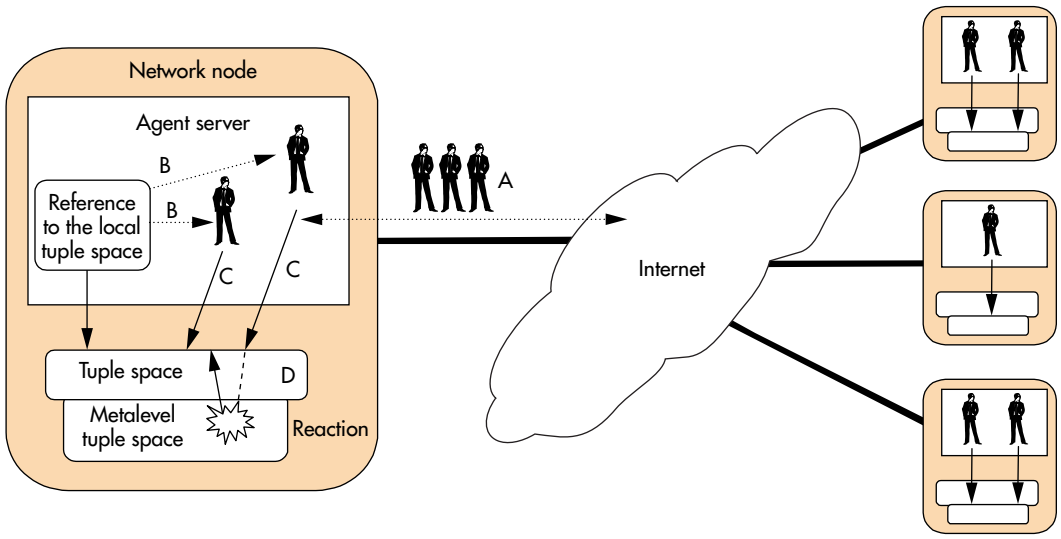

Figure 1. The MARS architecture. Agents arrive at the site (A) and are provided with a reference to the local MARS tuple space (B). They access the tuple space (C), which can react with a programmed behavior (D).

agent and agent-to-execution environment interactions depend on the tuple spaces' built-in data-oriented pattern-matching mechanism, which may not be suitable. It may be difficult, for example, to realize and control complex interaction protocols with just Linda-like pattern matching, and Linda's data-oriented interaction model does not provide an easy way to access a site's services.

Programmable tuple space models—where tuple-space access events trigger certain computational activities—solve these problems.[5,15] This reactivity offers several advantages for mobile agent applications:

- A site administrator can monitor access events and implement specific security policies for the execution environment.
- Tuples can be dynamically produced on demand, enabling a simple data-oriented mechanism for accessing a site's services. For example, an agent's attempt to read a specific tuple can trigger the execution of a local service that produces it.
- Application-specific coordination rules can be imposed.

One drawback of programmable tuple spaces is that irrational programming can distort application interactions and cause users to lose control over agent execution. Rather than altering application semantics, however, site administrators are more likely to be interested in exploiting the programmability of their tuple spaces to increase availability of data and resources while better protecting them. Also, if application-specific programming is properly constrained to only affect the application itself, agents from other applications will not experience unexpected tuple space behavior.

MARS COORDINATION ARCHITECTURE

MARS implements a portable and programmable Linda-like coordination architecture for Java-based mobile agents. It does not implement a whole new Java agent system, but instead complements existing agent systems. Because it is not bound to any specific implementation, MARS can be associated with different Java-based mobile agent systems with only slight modification. The current implementation, available for downloading at http://sirio.dsi.unimo.it/MOON, has already been tested with Aglets, Java-to-go, and SOMA.

As shown in Figure 1, MARS consists of many independent tuple spaces. Each tuple space is associated with a node and is accessed by locally executing agents. Integrating MARS with a mobile agent system therefore requires only that the agent server—which accepts and executes incoming agents on a node (step A in Figure 1)—supply agents with a reference to the local MARS tuple space (step B). Agents on a node can then access the local tuple space through a well-defined set of Linda-like primitives (step C), and each MARS tuple space can react to accesses with behaviors programmed in a metalevel tuple space (step D).

Other Programmable Coordination Architectures

Other agent-based Internet architectures besides JavaSpaces exploit tuple spaces. PageSpace, an architecture for interactive Web applications, uses Linda-like coordination.[1] Both mobile and fixed agents can use its tuple spaces to store and associatively retrieve object references. Also, agents can create tuple spaces to interact privately without affecting host execution environments.

To influence the coordination activities of application agents, PageSpace is not reactive in itself but instead defines special-purpose agents to access the space and change its contents. However, these special-purpose agents are very limited, both in monitoring interaction events and in tuning their effects, compared with MARS.

The T Spaces project at IBM uses Linda-like interaction spaces as general-purpose information repositories for networked and mobile applications.[2] Rather than defining agent-oriented coordination media, the T Spaces architecture aims to provide a powerful and standard interface for accessing large amounts of information organized as a database.

Therefore, the designers of T Spaces integrated a special programmability to add new behaviors to a tuple space. This occurs through new admissible operations on a tuple space—typically complex queries—rather than by programming the effects on the basic Linda operations, as does MARS. In our opinion, this makes T Spaces less usable in the open Internet environment because it requires application agents to either be aware of operations available in a given tuple space or somehow dynamically acquire this knowledge.

What further distinguishes MARS from these two systems is the way agents refer to tuple spaces. In MARS, agents hold a single reference that is implicitly bound to the tuple space of the local execution environment. But in PageSpace and T Spaces (as well as JavaSpaces), agents can refer to multiple tuple spaces, whether local or remote, using different tuple space references, accessing them without regard to their location. This can make managing applications more difficult, since the agent code has to explicitly manage tuple space references, and it can make agent execution less controllable and less reliable, since interactions occur transparent to the location of agents and tuple spaces.

The Tucson model deals with this problem by making agents refer to tuple spaces through URLs, as an Internet service, thus enforcing network awareness.[3] With regard to the tuple space model, Tucson resembles MARS in allowing full programming 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 complements MARS and makes Tucson well suited for application development based on intelligent information agents, which are in charge of accessing and managing large and heterogeneous information sources.

The MOLE system defines a meeting-oriented coordination model rather than a tuple-based one.[4] MOLE meetings use shared, nonmobile objects that agents must access to send or receive messages. Unlike Telescript and Ara meetings, however, MOLE meetings enforce temporal uncoupling—by permitting asynchronous message notification to agents—and can be programmed to integrate specific policies for managing the exchanged messages. These characteristics provide 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 definition of meeting points at the application level.

References

1. P. Ciancarini et al., "Coordinating Multi-Agents Applications on the WWW: a Reference Architecture," *IEEE Trans. Software Eng.*, vol. 24, no. 8, May 1998, pp. 362-375.
2. P. Wyckoff et al., "T Spaces," *IBM Systems J.*, vol. 37, no. 3, 1998, pp. 454-474.
3. A. Omicini and F. Zambonelli, "Coordination for Internet Application Development," *J. Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 3, Sept. 1999, pp. 251-269.
4. J. Baumann et al., "Mole—Concepts of a Mobile Agent System," *World Wide Web J.*, vol. 1, no. 3, 1998, pp. 123-137.

The MARS tuple space enables interagent communication and allows agents to access primitive data items and references to execution environment resources. The local tuple space is the only node resource that agents can directly access, reducing the problem of dynamically binding local references to mobile entities.[1]

MARS Interface

MARS complies with the JavaSpaces specification, likely to be Java's de facto tuple space interface. As in JavaSpaces, MARS tuples are Java objects whose instance variables represent tuple fields. Each tuple field is a reference to an object that can also represent a primitive data type (*wrapper objects* in Java terminology). Any tuple field can have either a defined (actual) value or a null (formal) value.

Each MARS tuple space is realized as an instance of the Space class, which implements the MARS interface, an extension of the JavaSpace interface.

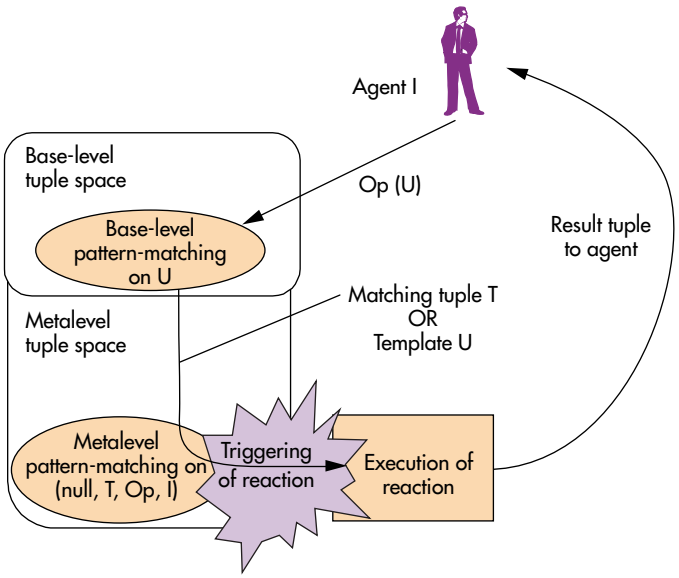

Figure 2. MARS metalevel activities. When an agent performs an access to the base-level tuple space, matching reactions are searched for in the metalevel tuple space and, if found, are executed.

To access the tuple space, the MARS interface provides the following operations:

- write, which puts a tuple, supplied as the first parameter, in the space;
- read, which retrieves a matching tuple from the space based on a template tuple;
- take, which works like read but extracts one matching tuple from the space;
- readAll, which retrieves all matching tuples from the space; and
- takeAll, which extracts all matching tuples from the space.

Tuple operations include transaction and timeout parameters. For example, the timeout parameter for read, take, readAll, and takeAll specifies how long operations wait for a matching tuple before returning a null value. A lease parameter sets a written tuple's lifetime.

The MARS interface adds the readAll and takeAll operations, which are not in the JavaSpaces interface, to prevent agents from having to perform several reads/takes to retrieve all needed information. This also avoids the risk of reading a tuple more than once, a well-known problem of tuple space models resulting from nondeterminism in the selection of a tuple among multiple matching ones.

The template tuple used with 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 following the object-oriented JavaSpaces pattern-matching rule.

Reactive Model

Linda's earliest definition provided a limited form of reactivity, the eval operation.[14] However, due to both the unclear semantics of eval and the lack of a suitable model for controlling tuple space computations, most implementations (including JavaSpaces) have omitted eval and other forms of reactivity. MARS, while also discarding eval, defines a flexible and controllable architecture for programming reactions to agent accesses to tuple spaces.

MARS reactions are stateful objects with a specific method implementing the reaction itself. Using metalevel 4-ples—tuples made up of four fields—stored in a metalevel tuple space, reactions match with specific access events according to three components: tuple item (T), operation type (Op), and agent identity (I). A metalevel 4-ple has the form (Rct, T, Op, I), meaning 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. When administrators (using the MARS graphical interface) or authorized agents insert or extract metalevel tuples, reactions associated with access events at the base-level tuple space are installed or uninstalled, respectively.

When a metalevel 4-ple has undefined values, it associates the specified reaction with all access events matching it. For example, the 4-ple (ReactionObj, null, read, null) in the metalevel tuple space associates the reaction of the ReactionObj instance with all read invocations, whatever the tuple type, content, or agent identity.

The metalevel tuple space follows associative mechanisms similar to those of the base-level tuple space. Any access to the base-level tuple space activates a metalevel pattern-matching mechanism, which detects reactions to trigger—metalevel tuples matching the access event—and executes them. For example, as shown in Figure 2, when an agent with identity I invokes a single-tuple input operation Op—read or take—supplying a template tuple U, the following occurs:

1. MARS issues the pattern-matching mechanism of the base-level tuple space to identify a tuple T that matches U.
2. MARS executes a readAll operation in the

metalevel tuple space by providing the 4-ple (null, T, Op, I), where T is the tuple T if matched in step 1 or the template U otherwise.
3. If MARS finds a matching 4-ple in the metalevel tuple space, it triggers the corresponding reaction. The reaction method receives the tuple T and the Op and I values as parameters and is expected to return a tuple.
4. If MARS does not find a matching metalevel 4-ple, it lets the invoked operation Op proceed according to normal semantics.

If several 4-ples satisfy step 2's matching mechanism, MARS executes all corresponding reactions in a pipeline—sequentially, one after another—in the order of their installation. Since execution order can affect the final result, MARS rules out nondeterminism for metalevel matches.

When an agent invokes a readAll or a takeAll in the base-level tuple space, multiple matches can occur during step 1 or during a reaction triggered in step 3. If multiple matches occur in step 1, MARS associates a pipeline of matching reactions with each matching tuple. If no matches occur in step 1, MARS activates a single reaction pipeline, and as soon as a reaction in the pipeline produces multiple tuples, MARS replicates the rest of the pipeline for each result tuple. For a write operation, MARS starts directly from step 2 and uses the tuple T, the parameter of the write operation itself, in the metalevel 4-ple (null, T, write, I).

Reactions can access the base-level tuple space and perform any operation on it, although to avoid endless recursion these operations do not themselves issue a reaction. Apart from the state in the reaction object, the base-level tuple space, then, can hold additional reaction state information. When the reaction executes, its parameters are

- the results of the matching mechanism issued by the associated operation, or by the template that invoked the input operation;
- the operation type; and
- the invoking agent's identity.

The reaction, on the basis of the actual tuple space content and of the past access event, can then influence the effects of operations in an informed way, for example returning to specific invoking agents tuples different from those of the normal pattern-matching mechanism.

These characteristics of the MARS reactive model allow great flexibility. This contrasts with the JavaSpaces notify mechanism, which simply signals external objects about tuple insertions in a tuple space, thus only monitoring write operations and not allowing control of the effects of tuple space operations.

Security Model

MARS security protects tuple spaces and their contents from unauthorized and malicious accesses. MARS assumes that the underlying mobile agent system identifies and authenticates agents. So, when an agent obtains the reference to the local MARS tuple space, MARS has already authenticated it and perhaps mapped it to a specific role.

Using a few well-defined agent roles has several advantages: First, it leads to a better uncoupling between the agent system and MARS. Second, roles can better handle the openness of mobile agent systems, where large numbers of possibly unknown identities will likely be encountered. Both agent authentication and role mapping let MARS simply control tuple space accesses by defining access control lists (ACLs) for tuple spaces and tuples and by referring to authenticated identities and roles.

MARS ACLs define who can do what on tuple spaces and their enclosed tuples, at both the base level and the metalevel. A tuple space system administrator can decide whether a specific agent or a specific role can perform a specific operation on a given tuple. An administrator can define, among others, three very general roles—reader, writer, and manager:

- *Reader* agents can read public tuples from the space but can neither insert nor extract tuples.
- *Writer* agents can both read and store tuples in the space but cannot extract tuples stored by agents with different identities.
- *Manager* agents, which are typically owned by administrators, have full rights to the space. They also have full access to the metalevel tuple space, and thus can dynamically install and uninstall reactions.

A tuple space administrator can define other roles to give or deny access to specific operations on specific tuple classes and to allow agents to install and uninstall reactions for a limited set of access events. In particular, site administrators should define proper role authorization policies to guarantee that external agents—by installing or uninstalling application-specific reactions—can influence only the activities of the agents of the same application.

```
. . .
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 < HTMLFile.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 3. Fragment of Java code for searcher agents.

MARS IN MOBILE AGENT APPLICATIONS

A good illustration of the MARS architecture's power is the searcher application, which sends mobile agents to remote Internet sites to analyze HTML pages and extract required information without having to transfer the pages over the network. For example, a searcher agent could go out on the Web and return with the URLs of pages containing a specific keyword.

To speed up the search, the application can create a tree of concurrent searcher agents. When one agent accessing an HTML page finds links to other potentially interesting pages, it clones itself, and the new agents follow the links, recursively continuing the search. In this case, coordination between the agent and the execution environment is needed to make agents access and retrieve information on a site, and coordination between agents is needed to avoid accessing a page already reviewed by another clone.

Coordination Between Agents and the Execution Environment

With client-server or meeting-oriented coordination, access to resources depends on the local server, but a server may not always provide appropriate services. For example, most Web servers do not provide site indexes, so agents must navigate page by page. With code mobility, agents can carry navigation code and install it on the sites visited, but this introduces security issues and requires exception handling. Also, agents must become control oriented, explicitly requesting services in order to obtain data and files.

But, with a blackboard or tuple space on sites, agents can access HTML pages without special services and in a more natural data-oriented style. With MARS, tuples in the local execution environment provide general file information—such as pathname, extension, and modification time—and contain a reference to a file object used to access file contents. Agents looking for HTML pages can obtain references to them by accessing the tuple spaces.

Figure 3 shows a fragment of the Java code for typical searcher agents. Agents invoke the readAll operation in nonblocking mode by providing a template FileEntry tuple in which only the Extension field is defined with the *html* value. The operation returns all matching tuples—those representing HTML documents—in a vector. For each matching tuple, the agent accesses the corresponding file using the ActualFile field and searches for the keywords of interest in its content. Finally, the agent searches for remote links in the files and clones itself to follow those links before going back home.

With MARS reactivity, however, the coordination between agents and the execution environment shown in Figure 3 can be made more flexible and secure without altering the agent code. For example, suppose a Web site uses *htm* file extensions instead of *html*. With reactivity, administrators could give access to agents requesting the more standard *html* extension without changing local file names, without providing FileEntry tuples in both *html* and *htm* forms, and without forcing agents to explicitly deal with the heterogeneity. Administrators simply need to install a reaction, such as the one shown in Figure 4, that changes any request for *html* tuples to one for *htm* tuples.

Coordination Between Agents

To avoid duplicated work, we need interagent coordination, but client-server coordination results in odd designs. Agents are dynamically created and spatially autonomous, so they are unaware of other agents. The only way for an agent to know whether others have already visited a site would be to invoke a specialized directory server, but this only complicates application design and, by requiring remote communication, makes it less efficient and reliable.

```
Class HTML2HTM 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

//no match already occurred if the site has only htm files
{(FileEntry)Fe.Extension = "htm"; //modifies the extension of the required files
     return s.readAll(Fe, null, NO_WAIT);
}}
```

Figure 4. The HTML2HTM reaction class. This reaction changes any request for html tuples to one for htm tuples.

Meeting-oriented coordination allows a more distributed solution, but this requires additional special-purpose entities. When an agent explores a site, it creates a meeting agent on the site to inform incoming agents of its visit. Meeting agents open meeting points as shared data spaces, so if the coordination model itself is based on shared data spaces—as are blackboards and tuple spaces—interagent coordination does not require peculiar designs.

With MARS, any searcher agent arriving at a site can check the local tuple space to see if another agent of the same application left a *marker tuple,* which has the form (my_application_id, "visited"). If one exists, the arriving agent terminates; if not, the agent does its work and leaves one marker tuple in the tuple space.

Searcher agents must be assigned a writer role in order to leave marker tuples, but this puts a site at risk of being overwhelmed with tuples no one will ever use or ever delete. MARS solves this problem by allowing administrators to install a reaction that specifies a site-specific allowed lifetime, disregarding any lifetime specified in the lease parameter at the application level.

A more sophisticated way to avoid duplicated work than using marker tuples is to let agents install application-specific reactions for access events performed by other agents of the same application. For example, agents could install a stateful reaction that avoids retrieval of duplicate information but also takes into account possible page updates. When an agent accesses the tuple space to retrieve HTML page references, the reaction could check for any matching FileEntry tuple and also determine whether the corresponding file has been modified since the last visit. If it hasn't been modified, the reaction simply does not return the corresponding tuple to the agent.

PERFORMANCE EVALUATION

The power and flexibility of the MARS architecture do not impose a high overhead on coordination activities. We have measured access times for a MARS tuple space, specifically evaluating overhead introduced by MARS. Figure 5 (next page) shows the time needed for 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, for different cases:

- by completely deactivating the metalevel activities (read—passive tuple space), which happens in any nonreactive tuple space implementation;
- by activating the metalevel activities, without any reaction installed (read—0 reactions issued—0 reactions installed); and
- by activating the metalevel 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).

Because more information is handled in the pattern-matching process, access times in all cases increased logarithmically with the number of tuples stored in the base-level tuple space (from 4 to 10 milliseconds in the tests performed).

Comparing access times with and without the metalevel matching mechanism activated shows that overhead introduced by MARS is very limited and independent of the global number of base-level tuples. When no reactions (metalevel tuples) have been installed, metalevel tuple space activities introduce overhead of about 10 percent, which grows very slightly as the number of installed reactions increases. For instance, in the unlikely case of 50

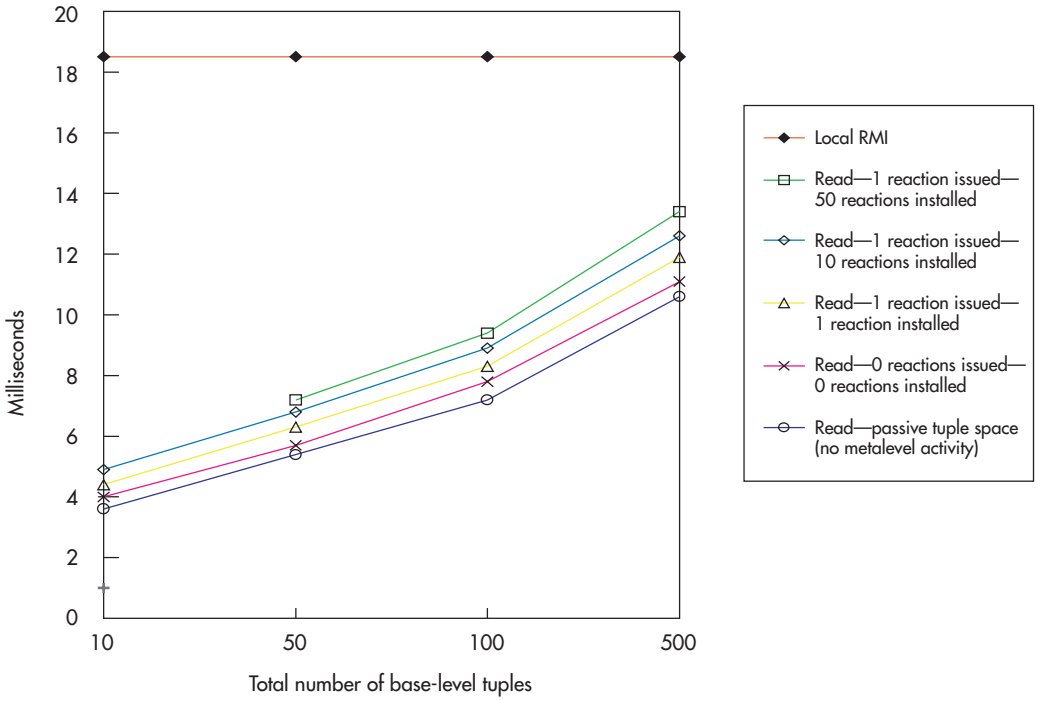

Figure 5. Overhead introduced by reactions. The graph shows the cost in time of the reactions in several cases, also compared with the nonreactive case.

installed reactions in the metalevel tuple space, where a read triggers a null-body reaction, overhead is still well below 30 percent. These results likely apply to other tuple space implementations, such as JavaSpaces, whenever they are extended with programmability.

The advantages of MARS-mediated interactions over other possible interaction mechanisms are also clear. For example, a null-body local remote method invocation (RMI) takes, on the same node, at least 18 ms, well over the average time for a MARS read. For remote interactions (not plotted in Figure 5), an RMI between a Sun Ultra 10 and a SPARCstation 5 connected through a 10-Mbyte Ethernet takes about 30 ms, which an Aglets messaging system increases to 37 ms. When you consider that the round-trip migration time for a 2.5-Kbyte Aglets agent in the same architecture is about 250 ms, it becomes clear that when an agent needs more than a dozen interactions with site resources, it is cheaper to send the agent to the site and let it interact locally through MARS.

CONCLUSION

Linda-like coordination models, enriched with tuple-space programming capabilities, are ideal for mobile agent applications, allowing simpler and more flexible application design. MARS defines a general and portable coordination architecture, which facilitates the design and development of Internet applications based on Java mobile agents.

We are now extending MARS for integration with a proxy-based framework for computer-supported cooperative work, which will lead to an open coordination environment for seamless interaction of agents and humans in the context of interactive Web applications.[12]

We are also trying to resolve some open problems associated with allowing foreign application agents to program tuple spaces. While MARS can now confine the effects of agent-installed reactions, it cannot deal with spamming of endless and computationally intensive reactions. We also need to develop garbage collection mechanisms to remove reactions that are no longer useful. These, however, are general problems of mobile agent technology, not problems specific to MARS, and they can be resolved by adding resource control mechanisms to the Java Virtual Machine and by defining resource accounting models for agents. ■

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.

Mobile Agents and Coordination Resources

You can find many of the mobile agent systems and coordination architectures mentioned in this article at the URLs listed below. Several sites offer free downloads of the systems.

Ambit • http://www.luca.demon.co.uk/Ambit/Ambit.html
ARA • http://www.uni-kl.de/AG-Nehmer/
D'Agents • http://www.cs.dartmouth.edu/~agent/
General Magic • http://www.genmagic.com/
IBM Aglets • http://www.trl.ibm.co.jp/aglets/
Java2go • http://ptolemy.eecs.berkeley.edu/dgm/javatools/java-to-go/
JavaSpaces • http://chatsubo.javasoft.com/products/javaspaces/
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/Research/TuCSoN/

REFERENCES

1. A. Fuggetta, G. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Trans. Software Eng.*, vol. 24, no. 5, May 1998, pp. 352-361.
2. N.M. Karnik and A.R. Tripathi, "Design Issues in Mobile-Agent Programming Systems," *IEEE Concurrency*, vol. 6, no. 3, July-Sept. 1998, pp. 52-61.
3. R.M. Adler, "Distributed Coordination Models for Client-Server Computing," *Computer*, vol. 29, no. 4, Apr. 1995, pp. 14-22.
4. D. Gelernter and N. Carriero, "Coordination Languages and Their Significance," *Comm. ACM*, vol. 35, no. 2, Feb. 1992, pp. 96-107.
5. G. Cabri, L. Leonardi, and F. Zambonelli, "Mobile-Agent Coordination Models for Internet Applications," *Computer*, vol. 33, no.2, Feb. 2000, pp. 82-89.
6. D.B. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets,* Addison-Wesley, Reading, Mass., 1998.
7. D. Kotz et al., "Agent TCL: Targeting the Needs of Mobile Computers," *IEEE Internet Computing*, vol. 1, no. 4, July-Aug. 1997, pp. 58-67.
8. J. White, "Mobile Agents," in *Software Agents*, J. Bradshaw, ed., AAAI Press, Menlo Park, Calif., 1997, pp. 437-472.
9. H. Peine, "Ara—Agents for Remote Action," in *Mobile Agents: Explanations and Examples*, W.R. Cockayne and M. Zyda, eds., Manning/Prentice Hall, Greenwich, Conn., 1997.
10. L. Cardelli and D. Gordon, "Mobile Ambients," *Foundations of Software Science and Computational Structures, Lecture Notes in Computer Science*, no. 1378, Springer-Verlag, Berlin, 1998, pp. 140-155.
11. P. Domel, A. Lingnau, and O. Drobnik, "Mobile Agent Interaction in Heterogeneous Environments," *First Int'l Workshop on Mobile Agents, Lecture Notes in Computer Science*, no. 1219, Springer-Verlag, Berlin, 1997, pp. 136-148.
12. P. Ciancarini et al., "Coordinating Multiagent Applications on the WWW: A Reference Architecture," *IEEE Trans. Software Eng.*, vol. 24, no. 8, May 1998, pp. 362-375.
13. E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces Principles, Patterns, and Practice,* Addison-Wesley, Reading, Mass., 1999.
14. S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *Computer*, vol. 19, no. 8, Aug. 1986, pp. 26-34.
15. A. Omicini and F. Zambonelli, "Coordination for Internet Application Development," *J. Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 3, Sept. 1999, pp. 251-269.

Giacomo Cabri is a PhD student in computer science at the University of Modena and Reggio Emilia. His current research interests include tools and environments for parallel and distributed programming and object-oriented programming. He is a member of the Italian Association for Object-Oriented Technologies (TABOO).

Letizia Leonardi is an associate professor in computer science at the University of Modena and Reggio Emilia. She received a PhD in computer science in 1988 from the University of Bologna. Her research interests include design and implementation of parallel object environments on distributed, massively parallel, and heterogeneous architectures. Leonardi is vice president of TABOO and a member of AICA.

Franco Zambonelli is a research associate in computer science at the University of Modena and Reggio Emilia. He received a PhD in computer science in 1997 from the University of Bologna. His current research interests include tools and environments for parallel and distributed programming, Internet computing, and coordination technologies. He is a member of ACM, EuroMicro, IEEE, AICA, and TABOO.

Readers can contact the authors at {giacomo.cabri, letizia.leonardi, franco.zambonelli}@unimo.it.