Autonomous Agents and Multi-Agent Systems, 2, 251–269 (1999)
© 1999 Kluwer Academic Publishers. Manufactured in The Netherlands.

Coordination for Internet Application Development

ANDREA OMICINI aomicini@deis.unibo.it
LIA—*DEIS—Università di Bologna, Bologna, Italy*

FRANCO ZAMBONELLI franco.zambonelli@unimo.it
DSI—Università di Modena, Modena, Italy

Abstract. The adoption of a powerful and expressive coordination model represents a key-point for the effective design and development of Internet applications. In this paper, we present the TuCSoN coordination model for Internet applications based on network-aware and mobile agents, and show how the adoption of TuCSoN can positively benefit the design and development of such applications, firstly in general terms, then via a TuCSoN-coordinated sample application. This is achieved by providing for an Internet interaction space made up of a multiplicity of independently programmable communication abstractions, called *tuple centres*, whose behaviour can be defined so as to embody the laws of coordination.

Keywords: coordination, Internet applications, mobile agents, programmable tuple spaces

1. Introduction

The design and development of Internet-based applications call for new paradigms, models, languages and tools, effectively supporting the vision of the Internet as a *globally distributed computing system*, where any kind of distributed information and resources can be globally accessed and elaborated.

Traditional distributed applications are designed as sets of entities statically assigned to a given execution environment, and cooperating in a (mostly) network-unaware fashion. However, enforcing transparency in a wide area network that lacks centralised control, as in the case of the Internet, is not a viable approach. A more suitable paradigm for the design of Internet applications is *network-aware computing* [14, 31]: the Internet is modelled as a collection of independent nodes, and applications are made up of network-aware entities (*agents*) which explicitly locate and access remote data and resources. Furthermore, the capability of the agents to proactively migrate to different execution environments (i.e., Internet nodes) during their execution (*agent mobility*) leads to a more efficient, dynamic and reliable execution scenario [31, 21]: mobile agents can move locally to the resources they need, requiring neither the transfer of possibly large amounts of data nor network connection during the access. From now on, we will globally refer to network-aware agents, either mobile or not, as *Internet agents*.

In the last few years, several systems and programming environments have appeared to support widely distributed applications based on Internet agents [22], by exploiting mobile code, security mechanisms and new Internet-oriented language constructs [14]. Surprisingly, a few of the proposals provide suitable mechanisms and abstractions for agent coordination. In our opinion, instead, the choice and exploitation of a suitable *coordination model* [17] is a key point in the design and development of Internet applications.

Most of the proposals rely on some forms of direct communication, based on peer-to-peer or client-server protocols, which we consider to be inadequate for Internet applications, since (i) direct communication between widely distributed, asynchronous and independent entities—as Internet agents are—is inherently difficult, (ii) direct communication is not expressive enough, *per se*, to deal with issues like heterogeneity, dynamicity, security, and so on. Tuple-based coordination models based on associative blackboards (like Linda [15]) seem to cope better with the Internet scenario, since they enable both spatial and temporal uncoupling in interaction, a feature which is essential in the case of widely distributed, autonomous and possibly mobile entities.

Still, the intrinsic data-orientation of Linda-like coordination models [29] makes them lack the flexibility (typical of control-oriented coordination models) required by the intrinsic dynamicity of the Internet: both agent-to-agent communication and agent access to local data are tied to the built-in associative mechanisms integrated in tuple spaces. Thus, any coordination policy not directly supported by the model (like atomically reading two tuples) has to be charged upon agents, which have to be made aware of the coordination laws. This increases the complexity of application design, by breaking the logical separation between coordination and algorithmic issues.

The above considerations motivate the definition of the TuCSoN (Tuple Centres Spread over Networks) model for the coordination of Internet agents, based on the notion of programmable coordination medium [9]. TuCSoN defines an interaction space made up of a multiplicity of independent communication abstractions, called *tuple centres*, spread over Internet nodes, and used by agents to interact with other agents as well as with each local execution framework. Tuple centres are enhanced tuple spaces, whose behaviour in response to agent triggered communication events is no longer restricted to the raw Linda-like pattern-matching mechanism but can be extended in order to embody specific coordination laws. This makes it possible to embed global system properties into the interaction space, by charging tuple centre behaviour with many issues critical to Internet applications, such as heterogeneity and dynamicity of the hosting environments, mobile agent cooperation over space and time, and incremental application development.

The rest of this paper is organised as follows. Section 2 discusses the issue of coordination for Internet agents. Section 3 describes the TuCSoN model. Section 4 presents an application example in the area of distributed information retrieval and discusses the impact of TuCSoN on it.

2. Motivation and related work

In agent-based Internet applications, both agent-to-agent and agent-to-execution environment interactions have to be modelled and managed. As argued in [2], the choice of the coordination model actually represents a key-point in the design process. In spite of that, among the several systems and programming environments aimed at supporting and facilitating the design and development of Internet applications [22, 21, 20], only a few of them focus on the definition of a suitable coordination model [7, 3].

2.1. *Direct interaction models*

In the context of mobile agent systems, *direct communication* between interacting components is usually adopted as the main interaction protocol. Java-based mobile agents systems [22, 20] adopt the client-server pattern typical of object-based systems, and can exploit the TCP/IP protocol at the socket level. In the context of intelligent multi-agent systems, the issue of inter-agent communication has been widely studied in the past few years [13, 19, 25], leading to the definition of sophisticated interaction models for knowledge exchange that are based on peer-to-peer direct communication.

In our opinion, the above approaches do not generally suit Internet agents. In fact, direct coordination implies a strict coupling between interacting entities in terms of name (who the partners are), place (where the partners are) and time (when the partners interact), which makes it difficult to deal with both agent mobility and execution environment unpredictability. In addition, mobility prevents direct inter-agent coordination from being scalable: while it is possible to let mobile agents interact in a small-sized network by tracking their movements, applying this method to the Internet would be expensive, ineffective and unreliable. Furthermore, direct inter-agent communication makes it difficult to enforce security policies, given that interaction typically occurs with no control by the hosting execution environments.

The above problems are partially solved by the *meeting-oriented* coordination model, implemented by the ARA [30] and MOLE [1] systems. In this model, interactions are forced to occur in the context of abstract meeting-points, implemented as special-purpose agents, which application agents join, either implicitly or explicitly, and can interact there with no need to know each other. However, agents still need to share the common knowledge of meeting points' identities and locations.

The interposition of specialised communication middleware—as in *CORBA* [26] and in *KQML* [13]—is an alternative approach to solving the identified problems of direct interaction models. However, apart from the uneasiness to scale to a world-wide area, middleware cannot help models based on direct communication in shaping the components' space of interaction according to the application needs. In fact, they do not provide the appropriate abstractions and metaphors needed to handle and rule the interaction space, which is actually what a *coordination model* is meant to do.

2.2. *Blackboard-based coordination*

Blackboard-based architectures [12] exploit shared data spaces (blackboards) to promote *indirect communication*. Since interaction is mostly ruled by the information exchanged rather than by the communication actions, blackboard-based coordination models are usually defined as *data-driven* ones [29].

By uncoupling interacting entities, blackboard-based coordination models solve many problems related to mobility and unpredictability: agents can interact via blackboards without knowing who and where the partners are. In addition, these models intrinsically provide a better support for security models, since blackboards (where all interactions occur) can easily be fully monitored. Furthermore, the scalability limit of computational models based on shared information spaces—that would made them inapplicable to the Internet area—can be easily overcome by enforcing locality: *local blackboards* can be associated to each node, to be used by agents to interact locally to their current execution environment.

Among several models, Linda tuple spaces [15, 17], based on *associative* blackboards, are particularly suitable to data-oriented applications. In fact, associative access permits agents to retrieve data from blackboards through some data-matching mechanism (like unification, or pattern-matching) integrated within the blackboards. This makes it easier to deal with data incompleteness as well as with dynamicity and heterogeneity of the information sources, which are typical of Internet nodes.

Some recent proposals recognise the suitability of blackboard-based coordination models in the context of Internet application, and define and implement locally shared information spaces. In Ambit [4], a recently proposed formal model for mobile computations, a mobile entity can "attach" a message to a given system, like a *post-it*, which another entity can retrieve and read later. However, no associative access to messages is provided. The ffMAIN agent system [11] defines mobile agents that interact with both other agents and the local resources of the hosting execution environment via an information space accessed through the HTTP protocol, which also integrates a simple associative mechanism. The PageSpace coordination architecture [7] defines multiple—Linda-like—information spaces where agents of any kind can store and associatively (i.e., by class name) retrieve object references.

2.3. *Toward programmable communication abstractions*

Despite the potential benefits of uncoupled and associative coordination [17], only a few proposals in the area of Internet agents already adopt associative communication spaces. In our opinion, this may be due to the fact that Linda-like coordination *lacks* flexibility and control: both agent-to-agent coordination and access to local data are bound by the built-in data-access mechanisms provided by the blackboard. Then, any coordination policy not directly supported by the model has typically to be charged upon agents. In an open, heterogeneous and unpredictable environment like the Internet, this is likely to notably increase the complexity of the agents, which are forced both to implement in their code the peculiar coordination protocols required by the application, and to solve Internet-related issues, such as heterogeneity and dynamicity of the information sources in the execution environments. In addition, forcing coordination rules to be distributed between blackboards and agents affects the global application design and breaks the logical separation between coordination and algorithmic issues.

In order to address this limit, two kinds of proposals have been made, which enhance the Linda communication kernel either (i) by permitting the addition of new communication primitives, or (ii) by permitting to program and extend the default behaviour of the primitives.

The former approach is adopted, for example, by T Spaces [23]: agents can add any new primitive to the tuple space, in order to implement any required transaction on the stored data. However, this approach cannot generally suit open applications, because exploiting the additional functionalities requires a strict coupling between the interacting entities, i.e., agents have to know which operations other agents may have installed in the tuple space.

The latter approach has been adopted in the past by a few (non Internet-oriented) systems like Law-governed Linda [24] and $\mathcal{ACLT}$ [27]. However, the capability of programming the behaviour of the communication abstractions in response to communication events suits Internet applications well, because agents can adopt the same communication primitives, independently of the fact that their default behaviour could have been extended. The TuCSoN coordination model for Internet-based mobile agents, presented in the following section, starts from the above considerations, and exploits the notion of *tuple centre*, an enhanced tuple space whose observational behaviour is not fixed once and for all by the coordination model, as in Linda, but can instead be tailored according to the specific application needs.

The MARS system [3], developed in the context of an affiliated research project, is (to the best of our knowledge) the only research proposal that defines and implements a programmable tuple space model for mobile agents. However, unlike TuCSoN, MARS exploits an object-oriented tuple space model, which makes it more suitable to service and network management applications.

The already mentioned PageSpace architecture [7] recognises the need to control and tune the interaction space. However, PageSpace relies on special-purpose agents associated to the interaction space to drive the local coordination laws, rather than on the programmability of the coordination media.

3. TuCSoN

TuCSoN (Tuple Centres Spread over Networks) is a coordination model for Internet applications based on network-aware and mobile agents. Following the concepts and terminology introduced in [28], TuCSoN's most relevant features, apart from its Linda-like coordination language, concern

— the TuCSoN *coordination space*, with its twofold interpretation as either a *global interaction space* made up of uniquely denoted communication abstractions, or as a collection of *local interaction spaces*, and
— the TuCSoN *coordination media*, that is, the communication abstractions whose behaviour can be defined so as to embed global coordination laws.

The first feature effectively supports the twofold role of Internet agents, as network-aware entities that locate and access Internet data and resources, and as roaming entities that transfer their execution on a site where they interact with local resources. The second one enriches the coordination model, which is basically data-oriented, with the flexibility and control required to deal with the complexity of Internet applications.

3.1. The coordination space

TuCSoN coordination space relies on a multiplicity of independent communication abstractions, called *tuple centres*, spread over Internet nodes and used by agents to interact with other agents as well as with the local execution framework. Each tuple centre is associated to a node and is denoted by a locally unique identifier. As shown by the example in Figure 1, each node provides its own version of the TuCSoN *(coordination) media space* (that is, the set of the admissible tuple centre identifiers), by virtually implementing each tuple centre as an Internet service.

As a result, each tuple centre can be identified via either its full Internet (*absolute*) name or its local (*relative*) name. More precisely, tuple centre *tc* provided by the Internet node *node* can be referred to by means of its absolute name *tc@node* from everywhere in the Internet, and by means of its relative name *tc* in the context of node *node*. According to the example in Figure 1, the name `access@lia.deis.unibo.it` univocally denotes the tuple centre `access` provided by the Internet node `lia.deis.unibo.it`, while the name `access` may denote one of the three `access` tuple centres provided by the three nodes, depending on the node where the name is used.

Correspondingly, the TuCSoN coordination space can be seen as providing for either a *global interaction space*, featuring a collection of uniquely denoted tuple

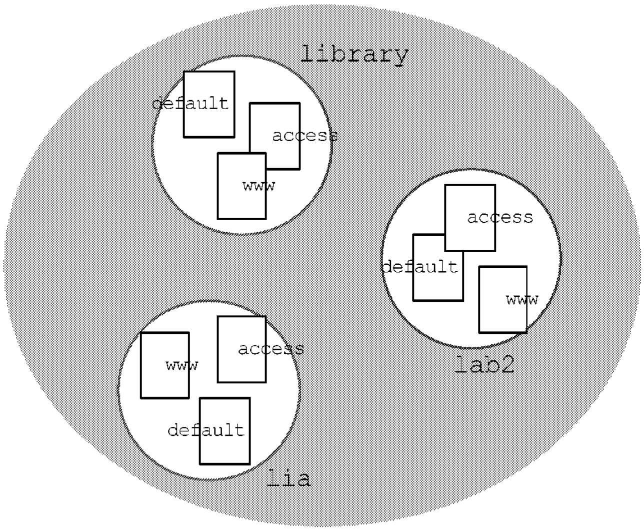

Figure 1. Three TuCSoN nodes {`lia, library, lab2`}`.deis.unibo.it`, each one implementing its local version of the same TuCSoN media space {`default, access, www`}.

centres (when referring to absolute tuple centre names), or a collection of *local interaction spaces*, defining the same set of identifiers (when referring to relative tuple centre names). For instance, the TuCSoN coordination space depicted in Figure 1 defines three distinct local versions of the same media space {`default`, `access`, `www`}. This space can also be interpreted as a unique, global media space {`default@lia.deis.unibo.it`, `default@library.deis.unibo.it`, ..., `www@library.deis.unibo.it`, `www@lab2.deis.unibo.it`}.

The following subsection should make clear how this can benefit Internet agent coordination, by simplifying the interaction protocol.

3.2. The coordination language

Agents interact by exchanging tuples via tuple centres by means of a small set of communication primitives (`out`, `in`, `rd`, `inp`, `rdp`) having basically the same semantics as Linda ones. According to [5], `out` writes a tuple in a tuple centre, while `in` and `rd` send a tuple template and expect the tuple centre to return a tuple matching the template, either deleting it or not from the tuple centre, respectively. The primitives `inp` and `rdp` work analogously to `in` and `rd`: however, while the latter wait until a matching tuple can be retrieved from the tuple centre, the former immediately fail if no such tuple is found.

The general form for any admissible TuCSoN communication operation performed by an agent is

tc?operation(tuple)

asking tuple centre *tc* to perform *operation* using *tuple*. Since *tc* can be either an absolute or a relative tuple centre name, agents can adopt two different forms of primitive invocation, the *network* and the *local* one, respectively, according to their contingent needs.

The network communication form *tc@node?operation(tuple)* is used by agents when they behave as network-aware entities, by denoting tuple centres through their absolute names in the global TuCSoN interaction space. For example, this form can be used by a mobile agent to remotely access a candidate hosting environment, query it, and possibly authenticate itself before migrating there.

The local communication form *tc@operation(tuple)*, referring by definition to the local tuple centre implementation of the current execution node of an agent, is used by agents when they behave as local components of their current hosting environment. This form is typically used by mobile agents to interact with local resources and to exchange data and knowledge with other agents.

The local communication form is not necessary, *per se*: an agent could always refer to a tuple centre via its absolute name, even when accessing the tuple centres of its current execution environment. However, the availability of both communication forms enforces the separation between network-related issues (such as agent migration across the nodes) and local computation issues (such as the interaction with the resources local to a node), thus reducing agent complexity. The above approach is somehow analogous to the one provided by most file systems, where pathnames can be specified in a relative (context-dependent) form, to facilitate file management activities.

3.3. The coordination media

As discussed in Subsection 2.3, a typical problem of Linda-like coordinated systems relies on the tuple space built-in and fixed behaviour: neither new primitives can be added, nor can new behaviour be defined in response to communication operations. As a result, either the support provided by the communication device is enough for the application purposes, or the interacting entities have to be charged in their code with the burden of the coordination. For this reason, TuCSoN exploits *tuple centres* as its coordination media, where a tuple centre is a tuple space enhanced with the notion of *behaviour specification*. More precisely, a tuple centre is a communication abstraction which is perceived by the interacting entities as a standard tuple space, but whose behaviour in response to communication events can be defined so as to embed the laws of coordination.

According to Subsection 3.1, each TuCSoN node provides for a multiplicity of tuple centres. Then, TuCSoN shares the advantages of models based on multiple tuple spaces (such as enhanced expressiveness and support for modularity, information hiding and security [16, 6]), and goes beyond, since the behaviour of every single tuple centre can be defined separately and independently of any other tuple centre according to specific coordination tasks. Thus, different communication abstraction can encapsulate different coordination laws, providing system designers with a finer granularity for the implementation of global coordination policies.

Generally speaking, the behaviour of a stateful communication abstraction like a tuple space is naturally defined as the observable state transition following a communication event. As a result, defining a new behaviour for a tuple centre basically amounts to specifying a new state transition in response to a standard communication event. This is achieved by enabling the definition of *reactions* to communication events via a *reaction specification language* [10]. More precisely, a reaction specification language makes it possible to associate any of the TuCSoN basic communication primitives (`out`, `in`, `rd`, `inp`, `rdp`) to specific computational activities, called reactions.

A reaction is defined as a set of non-blocking operations, and has a success/failure transactional semantics: a successful reaction may atomically produce effects on the tuple centre state, a failed reaction yields no result at all. Each reaction can freely access and modify the information collected in the form of tuples in a tuple centre, and can access all the information related to the triggering communication event (such as the performing agent, the operation required, the tuple involved,...). Each communication event may in principle trigger a multiplicity of reactions: however, all the reactions executed as a consequence of a single communication event are all carried out in a single transition of the tuple centre state, before any other agent-triggered communication event is served.

As a consequence, from the agents' viewpoint, the result of the invocation of a communication primitive is the sum of the effects of the primitive itself and of all the reactions it triggered, perceived altogether as a single-step transition of the tuple centre state. With respect to the default tuple space semantics, such a transition is no longer limited to being the simple one (such as adding/deleting one single tuple) determined once for all by the model, but can instead be made as complex as desired by the system designer. This makes it possible to uncouple the agent's view of the tuple centre (which is perceived as a standard tuple space) from the tuple centre actual state, and to connect them according to the application needs.

3.4. *Tuple centres with* ReSpecT

The TuCSoN model for the coordination of mobile agents abstracts from both the communication language (tuple kind and matching criterion) and the language adopted for tuple centre behaviour specification. In this paper, we adopt the ReSpecT model [10] for TuCSoN tuple centres, defining both the communication and the specification language, since (i) it is already well-established from both a theoretical and an implementation viewpoint, and (ii) it provides communication with the expressiveness of first-order logic, enabling the twofold interpretation of the communication abstraction as both a message repository and a theory of the interaction [27].

A ReSpecT tuple centre contains (ordinary) tuples and *specification tuples*. A ReSpecT tuple is a first-order logic ground unitary clause, for which the Prolog syntax is adopted. The collection of the logic tuples of a ReSpecT tuple centre constitutes its *tuple space* part. A ReSpecT specification tuple, instead, is a Prolog unitary clause of the form `reaction(Op, R)`, which associates a communication event represented as a logic term `Op` to the reaction `R`. The collection of all the specification tuples of a tuple centre constitutes its *specification space* part, i.e., the tuple centre *behaviour specification*.

Given a tuple centre `tc` and its behaviour specification S_{tc}, the occurrence of a communication event E (such as the incoming request to `tc` to perform an `out` operation) triggers *all* the reactions R such that a specification tuple `reaction(Op, R)` $\in S_{tc}$ exists, and E matches `Op` (that is, the representation of E as a logic term `E` unifies with `Op`). All the reactions triggered are executed according to the transactional semantics defined in the previous subsection for any TuCSoN tuple centre.

Each ReSpecT reaction is syntactically defined as a conjunction of *reaction goals*, which can (i) access the information related to the triggering communication event, (ii) manipulate terms, and (iii) read, write and consume tuples. In particular, predicates like `out_r`, `in_r`, `rd_r`, and `no_r` make it possible to access and modify the space of the tuples. More precisely, `out_r` basically works as a conventional `out`, while `in_r` and `rd_r` have the same effect as `inp` and `rdp`, respectively. Complementary to `rd_r`, `no_r` succeeds when its argument tuple does not unify with any tuple in the tuple space, but fails otherwise.

Further ReSpecT reaction predicates are related to the semantics of some communication primitives. First of all, `in` and `rd` may be seen as made up of two distinct communication events [18]: the first query phase (*pre* phase), when a tuple template is provided, and the subsequent answer phase (*post* phase), when a matching tuple is eventually returned to the querying agent. According to that, ReSpecT introduces the two `pre/0` and `post/0` predicates, succeeding only in the *pre* and *post* phase, respectively, so that one may define reactions succeeding only in the *pre* or *post* phase of `in`/`rd` operations. Analogous considerations apply to `inp` and `rdp`, too. However, since these primitives may either succeed or fail, ReSpecT introduces two further predicates, `success/0` and `failure/0`, which succeed only in the case that the current non-blocking primitive succeeded or failed, respectively. This makes it possible to define different reactions for the *post* phase of an `inp` or a `rdp`, depending on the success or failure of the operation.

Reactions can be defined not only for communication primitives, but also for operations on tuples performed inside reactions. As a result, in a `reaction(`*`Op`*`, `*`Body`*`)` tuple, *`Op`* may be not only `out(`*T*`)`, `in(`*T*`)`, `rd(`*T*`)`, `inp(`*T*`)`, or `rdp(`*T*`)`, but also `out_r(`*T*`)`, `in_r(`*T*`)`, `rd_r(`*T*`)`, or `no_r(`*T*`)`, where *T* stands for a logic tuple.

As a simple example, consider the following reaction:

```
reaction(out(p(_)), (in_r(p(a)), in_r(p(X)), out_r(pp(a, X))))
```

which is triggered whenever a new logic tuple with predicate `p` and one argument is inserted in the tuple centre by an `out`. Its intended effect is to replace two `p` tuples, each one with one argument (one of which has to be `a`) with a single `pp` tuple with the two arguments of the `p` tuples.

Since the reactions to a communication event are executed only after the event has actually occurred, the `p` tuple emitted is already in the centre space when reaction execution starts. If even a single reaction goal fails (possibly because there is no further `p` tuple in the tuple centre, apart from the one just inserted by the `out`), the whole reaction fails, and its execution yields no effects at all on the tuple centre. If the reaction succeeds, instead, all its associated side-effects (removal of two `p` tuples, and insertion of one `pp` tuple) are realised altogether as a single state transition of the tuple centre. For instance, if tuple centre `tc` contains only the ordinary tuple `p(a)` and the specification tuple above, the invocation of `tc?out(p(b))` actually results in changing the set of the tuples of `tc` from {`p(a)`} to {`pp(a,b)`} (instead of {`p(a),p(b)`}, as expected in the case of a standard tuple space).

4. Building an Internet application with TuCSoN

This section is aimed at showing the impact of the TuCSoN coordination model on the design and development of Internet applications, by discussing a simple example which raises several typical problems of Internet application development: heterogeneity of the information sources, incremental specification and development, safe and secure access to resources.

Let us consider an Internet-based heterogeneous information system consisting of a collection of WWW servers, which differ in both their software and hardware architectures, as well as in the way in which knowledge is organised and represented.

Think for instance of a group of servers belonging to different research groups, containing information about their research activities which the groups mean to share. In order to build a global information service from this heterogeneous collection, the servers federate and constitute a TuCSoN environment: each federated node implements TuCSoN as an Internet service, and provides its local version of the same TuCSoN name space.

We suppose that an application has to be set up in this framework, exploiting mobile agents to gather information about knowledge organisation on the WWW servers, and automatically producing HTML pages of references on its home site. In this context, mobile agents are used as 'information retrievers' roaming all the federated servers, gathering all relevant information, and coming back to their home site.

4.1. Application design

The design of a multi-component application should define the interacting entities and their goals, as well as their interaction protocol, which determines (i) their observational behaviour, and (ii) their perception of the outside world. In this context, exploiting a coordination model means shaping the interaction space according to the abstractions provided by the model.

In our application example, mobile agents (acting as information retrievers roaming a collection of federated WWW servers) and local applications (acting as information sources hosted by a single WWW server) interact via a multiplicity of ReSpecT tuple centres, constituting the TuCSoN interaction space.

Given one or more keywords, each agent has to retrieve the URL of every HTML page concerning each keyword. Then, TuCSoN can be exploited at its best by ensuring that:

— Each federated WWW server provides a tuple centre named `www` recording the server hypertextual structure, as well as the knowledge contained.
— Each `www` tuple centre is programmed on each WWW server so that it is perceived by the agents as a tuple space containing one tuple of the form `kwURLs(`*`KW`*`,`*`URLs`*`)` for any possible keyword *`KW`*, where *`URLs`* is the list of all the URLs of the pages concerning *`KW`*, coherently with the current state of the server.

This makes it possible to design agents around a very straightforward protocol, such as the one sketched in Figure 2 with a sample imperative pseudo-language. From any hosting node, the agent first interacts remotely (as a network-aware entity) with the nodes it needs to explore, using the network communication form, then it moves there (if allowed). In the new hosting node, the agent interacts with the host's resources through the local media space using the local communication form. What Figure 2 helps to point out is that the interaction with local resources is always performed in the same way, independently of the current hosting node: wherever it is currently located, the agent simply reads tuples of the form `kwURLs(`*`KW`*`,`

```
...
mySelf = "Explorer Agent";keyword="TuCSoN";
while(nextNode = head(NodeList)) {
    // ask access to the nextNode host
    // with the network communication form
    if (access@nextNode?rdp(authorised(id(mySelf)))) {
        // if access is granted migrate to nextNode
        moveTo(nextNode);
        // locally access the tuple centre library
        // of the hosting node
        // with the local communication form
        www?rd(kwURLs(keyword, ?URLs));
        elaborateAndRecord(URLs);
    }
    NodeList = tail(NodeList);
}
...
```

Figure 2. A simple interaction protocol for a TuCSoN mobile agent.

URLs) from the local version of the `www` tuple centre. For example, when looking for all the pages related to the TuCSoN project, the agent would perform a `www?rd(kwURLs("TuCSoN",URLs))` invocation, expecting as a result the instantiation of variable *URLs* to the list of all URLs of the HTML pages of the server containing some references to TuCSoN.

4.2. *Heterogeneity*

The issue of the heterogeneity of information sources may be faced in principle by making mobile agents aware of all the different ways in which knowledge is represented and organised. This choice, however, would make agent design a highly complex task, and produce a very inflexible structure. For instance, in our application example, this choice would possibly imply the re-design of agents when a new federated server is added.

In TuCSoN, instead, the burden of heterogeneity can be charged upon tuple centres, which bridge the gap between the agent interaction protocols and the particular knowledge representation model adopted by each site. Take for instance two federated WWW sites, *A* and *B*, recording page-content relations in two different ways.

Server *A* describes the content of each page in terms of keywords by means of tuples of the form `keyword(`*KW*`, `*PageName*`)`, while page organisation is recorded by means of tuples of the form `page(`*PageName*`, `*PageURL*`)`. Both kind of tuples are stored in the local `www` tuple centre. For instance, the logic tuples `keyword("TuCSoN", "TuCSoN Home")` and `page("TuCSoN Home",`

"/TuCSoN/index.html") in the tuple centre www relate the location of the TuCSoN home page with the keyword "TuCSoN".

In order to be perceived by agents as a tuple space containing one tuple kwURLs(*KW, URLs*) for any possible keyword *KW*, the tuple centre www provided by server A is programmed to react to an rd(kwURLs(*KW, URLs*)) operation (i) by checking whether the required tuple is available, (ii) if not, by finding all the pages referring to keyword *KW*, (iii) by building the list of their corresponding URLs, and (iv) by finally providing the agent with the required answer tuple.

For instance, if the tuple centre www local to A contains the tuples

```
{  keyword("TuCSoN","TuCSoN Home"),
   keyword("TuCSoN","ReSpecT Home"),
   page("TuCSoN Home","/TuCSoN/index.html"),
   page("ReSpecT~Home","/ReSpecT/index.html")  }
```

then the invocation

```
www?rd(kwURLs("TuCSoN",URLs))
```

makes www react by adding tuple

```
kwURLs("TuCSoN",["/TuCSoN/index.html","/ReSpecT/index.html"])
```

to its tuple set. The corresponding ReSpecT code is presented in Figure 3a.

In its turn, server *B* is built around a DBMS application recording the site content. The DBMS interacts with the tuple centre www through a wrapper, translating tuples into queries, and answers into tuples. The wrapper waits for tuples of the form query(*Query*) in www and translates them into queries for the DBMS. Then, it waits for the DBMS answer and translates it into a tuple answer(*Query, TableList*), where *TableList* is the answer table provided in the form of a list, which is put in tuple centre www. In particular, a query of the form query(kwSearch(*KW*)) makes the wrapper ask the DBMS to return the URLs of all the pages containing references to keyword *KW*. The consequent answer is then given as a tuple answer(kwSearch(*KW*), *URLs*) in www.

Tuple centre www of the server *B* can be programmed to react to an incoming rd(kwURLs(*KW, URLs*)) invocation by producing a tuple query(kwSearch(*KW*)) for the wrapper. When the corresponding tuple answer(kwSearch(*KW*), *URLs*) is inserted by the wrapper in www, the tuple kwURLs(KW, URLs) initially required by the agent is finally produced.

For instance, the invocation of www?rd(kwURLs("TuCSoN", URLs)) on server *B* generates tuple query(kwSearch("TuCSoN")), which is consumed by the wrapper and translated into the proper query to the DBMS. If the DBMS returns a table with the two entries "/TuCSoN/index.html", "/MobAg/index.html" as its answer, the wrapper inserts the tuple answer(kwSearch("TuCSoN"), ["/TuCSoN/index.html", "/MobAg/index/html"]) in www. In the end, the inserted tuple is transformed into the tuple kwURLs("TuCSoN", ["/TuCSoN/index.html", "/MobAg/index.html"]) required by the agent. The corresponding code is shown in Figure 3b.

```
reaction(rd(kwURLs(KW,_)), ( pre,                 % When a kwURLs(KW,_) tuple is read,
  no_r(kwURLs(KW,_)),                             % but it is unavailable,
  out_r(kwPages(KW,[])) )).                       % start building the list of pages on KW
reaction(out_r(kwPages(KW,Pages)), (              % For any keyword(KW,Page) tuple,
  in_r(keyword(KW,Page)),                         % add Page to the page list
  out_r(kwPages(KW,[Page|Pages])) )).             % and keep on building the list
reaction(out_r(kwPages(KW,Pages)), (              % Stop building the list of pages on KW:
   no_r(keyword(KW,_)),                           % no more keyword(KW,_) tuples are left, so
   out_r(pageURLs(KW,Pages,[])) )).               % start building the list of the URLs
reaction(out_r(kwPages(KW,Pages)),                % Delete the kwPages/2 tuple
  in_r(kwPages(KW,Pages))).                       % just outed
reaction(out_r(pageURLs(KW,[Page|Pages],URLs)), ( % For any Page found,
  rd_r(page(Page,URL)),                           % get the corresponding URL,
  out_r(keyword(KW,Page)),                        % restore the previously consumed keyword/2 tuple
  out_r(pageURLs(KW,Pages,[URL|URLs])) )).        % and keep on building the URL list
reaction(out_r(pageURLs(KW,[],URLs)),             % When no more Pages are left, produce
  out_r(kwURLs(KW,URLs))).                        % the kwURLs(KW,URLs) tuple initially required
reaction(out_r(pageURLs(KW,Pages,URLs)),          % Delete the pageURLs/3 tuple
  in_r(pageURLs(KW,Pages,URLs))).                 % just outed
```

Figure 3a. $S_{\mathtt{www}@A}$: behaviour specification for the `www` tuple centre of server *A*.

```
reaction(rd(kwURLs(KW,_)), ( pre,            % Intercept a rd(kwURLs(KW,_)) operation
  no_r(kwURLs(KW,_)),                        % when no kwURLs(KW,_)tuple is available, and
  out_r(query(kwSearch,KW)) )).              % translate it into a query tuple for the DBMS
reaction(out(answer(kwSearch(KW),URLs)), (   % Get the answer tuple from the DBMS,
  in_r(answer(kwSearch(KW),URLs)),           % delete it from the tuple centre, and
  out_r(kwURLs(KW,URLs)) )).                 % translate it into the kwURLs/2 tuple required
```

Figure 3b. $S_{\mathtt{www}@B}$: behaviour specification for the `www` tuple centre of server *B*.

4.3. Application incremental development

Today complex software systems typically grow and change over time, to dynamically embody new features and provide new services, so that their design should intrinsically support and promote their incremental development. Since they are the core of a flexible design process, TuCSoN tuple centres can be easily exploited so as to take charge of the burden of incremental development, which can be performed transparently to the already existing components.

Let us assume that the federated WWW server *C* has the same structure and organisation as the server *A* described in the previous subsection. Then, suppose that an intelligent agent *L* is added to *C*, to learn from user interaction and infer new information from the user's exploration paths. In particular, say that *L* is able to infer that two knowledge items are strictly related from a semantic viewpoint. So, whenever such a new relation is inferred, *L* makes it available to the world by emitting a tuple of the form `relKW(`*`KW, RKW`*`)` in the tuple centre `www`, stating that whichever refers to *`RKW`* typically refers to *`KW`* too. For example, if *L* infers that pages concerning coordination typically refer to TuCSoN, too, it adds the tuple `relKW("TuCSoN","coordination")` to `www`.

It would then be desirable to integrate the new component *L* in the whole system in a transparent way, by making knowledge inferred by *L* available to explorer agents without affecting their interaction protocol. To this end, a TuCSoN tuple centre can be exploited to let a mobile agent, which is moving to *C* and asking for keyword *`KW`*, transparently get all the pages concerning both *KW* and all its related keywords, as inferred by L so far. Thus, whenever a new `relKW(KW, RKW)` tuple is inserted by L, `www` is programmed to react by looking for all the pages referring to *RKW*: for each tuple `keyword(RKW, PageName)` found, a tuple `keyword(KW, PageName)` is added to `www`, stating that the page concerns *KW*, too. Since any information previously produced about *KW* in the form of a `kwURLs(KW, URLs)` tuple is now out of date, it is finally removed, so that the first subsequent request by an agent causes the production of a tuple `kwURLs` containing the updated information.

For instance, if the tuple `relKW("TuCSoN", "coordination")` is put into `www` when `keyword("coordination", "Coordination Home")` is already in, reactions add a tuple `keyword("TuCSoN", "Coordination Home")`. Subsequently, if `www` already contains tuple `kwURLs("TuCSoN", ["/TuCSoN/index.html", "/ReSpecT/index.html"])`, it is removed, so that any subsequent request for such a tuple results in the insertion of the tuple `kwURLs("TuCSoN", ["/TuCSoN/index.html", "/ReSpecT/index.html", "/coordination/index.html"])`. Figure 4 shows the corresponding ReSpecT code. It should be noted that the behaviour specification of the tuple centre `www` local to C is simply given by the union of the specification tuples in Figure 3a with those in Figure 4, that is, more formally,

$$S_{\mathtt{www}@C} = S_{\mathtt{www}@A} \cup \Delta S_{\mathtt{www}@C}$$

This shows how TuCSoN makes the process of incremental development straightforward.

4.4. *Safety and security*

Malicious or simply badly programmed agents may undermine the internal coherence of the hosting execution environments. A knowledge source should then be

```
reaction(out(relKW(KW,RKW)),                     % When new information is inferred by the learner
   out_r(newPages(KW,RKW,[]))).                  % start the process to find new related pages
reaction(out_r(newPages(KW,RKW,Pages)), (        % Collect all the Pages concerning RKW:
   in_r(keyword(RKW,Page)),                      % if another Page exists,
   out_r(newPages(KW,RKW,[Page|Pages])) )).      % add it to the list
reaction(out_r(newPages(KW,RKW,Pages)), (        % Collect all the Pages concerning RKW:
   no_r(keyword(RKW,_)),                         % if no Page is left,
   out_r(relPages(KW,RKW,Pages)) )).             % state that Pages concern both KW and RKW
reaction(out_r(newPages(KW,RKW,Pages)),          % Delete the newPages/3 tuple
   in_r(newPages(KW,RKW,Pages))).                % just outed
reaction(out_r(relPages(KW,RKW,[Page|Pages])),(  % Until there are Pages,
   out_r(keyword(KW,Page)),                      % state they concern KW
   out_r(keyword(RKW,Page)) )).                  % as well as RKW
reaction(out_r(relPages(KW,RKW,[])),             % When no more Pages are left, if it is available,
   in_r(kwURLs(KW,_))).                          % delete the kwURLs(KW,_)) tuple, now out-of-date
reaction(out_r(relPages(KW,RKW,Pages)),          % Delete the relPages/3 tuple
   in_r(relPages(KW,RKW,Pages))).                % just outed
```

Figure 4. $\Delta S_{\mathtt{www}@C}$: further specification tuples for the `www` tuple centre of C.

```
reaction(out(updatePage(Name,NewURL)), (    % If an updatePage(Name,NewURL) tuple is outed,
   in_r(updatePage(Name,NewURL)),           % delete it immediately,
   in_r(page(Name,_)),                      % delete the old location information of Page
   out_r(page(Name,NewURL)) )).             % and write the updated one
reaction(out(page(PageName,URL)),           % If a pageName/2 tuple is outed,
   in_r(page(PageName,URL))).               % delete it immediately
reaction(in(page(_,_)), ( post,             % If a pageName/2 tuple is consumed by an in,
   current_tuple(page(PageName,URL)),       % determine its arguments
   out_r(page(PageName,URL)) )).            % and replace it immediately
reaction(inp(page(_,_)), ( post, success,   % If a pageName/2 tuple is consumed by an inp,
   current_tuple(page(PageName,URL)),       % determine its arguments
out_r(page(PageName,URL)) )).               % and replace it immediately
```

Figure 5. $\Delta S_{\mathtt{www}@A}$: further specification tuples for the www tuple centre of A.

protected from those interactions which could affect the semantic consistency of the information contained. In a coordination model like TuCSoN, where communication is mediated by a shared data space, the global consistency of a system with respect to interaction can be granted by providing for the consistency of the communication state (the space of the tuples, in TuCSoN).

Take again the server A sketched in Subsection 4.2, where page organisation is recorded through tuples of the form page(*PageName, PageURL*) contained in the local version of the tuple centre www. Then, we may like to ensure that, at any time, each HTML page of the server has its own unique URL recorded in www. This means that, given a *PageName* page, it should never happen that a *page(PageName, PageURL)* tuple is not present in www, and, dually, that two or more tuples of that kind occur in www at the same time.

The behaviour of tuple centre www can then be defined so as to avoid insertions and removals of *page(PageName, PageURL)* tuples, by adding to www behaviour specification reactions that make any in, inp, or out operation involving such tuples ineffective. Coherently, www behaviour can be programmed to make the atomic update of a page location possible through the emission of an updatePage(*PageName, NewURL*) tuple. The corresponding reactions (i) delete the updatePage tuple just inserted in www, (ii) delete the page tuple recording the old page URL, (iii) add the page tuple recording the new page URL. These three steps are performed altogether atomically in a single transition of the tuple centre state, thus preserving information consistency at the agent's perception level.

For example, if tuple centre www contains the tuple page("TuCSoN Home", "/TuCSoN/index.html"), the local invocation of a call www?out(updatePage ("TuCSoN Home", "/TuCSoN/home.html")) atomically results in the removal of page("TuCSoN Home", "/TuCSoN/index.html") and in the addition of page ("TuCSoN Home", "/TuCSoN/home.html") to www. If we denote with A' the server A once enriched with the ReSpecT reactions described in this subsection (denoted as a whole as $\Delta S_{\mathtt{www}@A}$ in Figure 5), the behaviour specification of A' is then given by

$$S_{\mathtt{www}@A'} = S_{\mathtt{www}@A} \cup \Delta S_{\mathtt{www}@A}$$

5. Conclusions and future work

The most promising approach to the design and development of distributed applications over the Internet is based on network-aware and mobile agents. However, this calls for new models and languages to manage interactions among the application agents.

The paper identifies the main features of a coordination model suitable for Internet applications and presents the TuCSoN coordination model, based on independently programmable communication abstractions local to each node, called tuple centres. By defining their behaviour in response to communication events, tuple centres can be charged with many issues critical to Internet applications, such as heterogeneity and unpredictability of the execution environments, and (mobile) agent cooperation over space and time.

As shown in this paper, the TuCSoN model can be effectively exploited in the context of distributed information retrieval. However, we have already identified several other application areas that could benefit from the programmable tuple space model defined by TuCSoN such as agent-mediated electronic commerce and distributed workflow management.

In spite of its many features, the TuCSoN model leaves some problems open. In fact, the real Internet structure is not a flat one, but can be considered to be a hierarchy of administrative domains separated by firewalls and gateways. So, we feel that TuCSoN should be able to model this kind of topology, with regard both to the naming of the tuple centres and to their local programmability. Furthermore, security issues, such as agent authentication and privacy, or tuple centre protection against malicious attacks, should be directly supported by the model. Our research is currently focusing on these topics that, though neglected by this paper, nevertheless play a key role in the context of Internet applications. The first result of our efforts is an extension of the TuCSoN model [8], which provides a unique, coherent framework where coordination is exploited as the basis for dealing with network topology, authentication and authorisation in a uniform way.

Acknowledgments

This work has been carried out under the financial support of the MURST (the Italian “Ministero dell’Università e della Ricerca Scientifica e Tecnologica”) in the framework of the Project MOSAICO “Design Methodologies and Tools of High Performance Systems for Distributed Applications.”

References

1. J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, and M. Strasser, “Communication concepts for mobile agent systems,” in *Mobile Agents 97*, LNCS 1219, Springer-Verlag, Berlin, 1997, pp. 123–135.
2. G. Cabri, L. Leonardi, and F. Zambonelli, “How to coordinate Internet applications based on mobile agents,” in *Proceedings of the 7th IEEE Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*, IEEE CS Press, 1998.

3. G. Cabri, L. Leonardi, and F. Zambonelli, "Reactive tuple spaces for mobile agent coordination," in *Proceedings of the 2nd Workshop on Mobile Agents*, LNCS 1477, Springer-Verlag, Berlin, 1998.
4. L. Cardelli and A. D. Gordon, "Mobile ambients," in *Foundation of Software Science and Computational Structures*, 1998.
5. N. Carriero and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. **32**(4), pp. 444–458, 1989.
6. P. Ciancarini, "Distributed programming with logic tuple spaces," *New Generation Computing*, vol. **12**, 1994.
7. P. Ciancarini, R. Tolksdorf, F. Vitali, D.Rossi, and A. Knoche, "Coordinating multiagent applications on the WWW: A reference architecture," *IEEE Transactions on Software Engineering*, vol. **24**(5), pp. 362–375, 1998.
8. M. Cremonini, A. Omicini, and F. Zambonelli, "Modelling network topology and mobile agent interaction: an integrated framework," in *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC '99)*, San Antonio, Texas, USA, February 28–March 2, 1999.
9. E. Denti, A. Natali, and A. Omicini, "Programmable coordination media," in *Coordination Languages and Models*, LNCS 1282, Springer-Verlag, Berlin, 1997, pp. 274–288.
10. E. Denti, A. Natali, and A. Omicini, "On the expressive power of a language for programming coordination media," in *Proceedings of the 1998 ACM Symposium on Applied Computing (SAC '98)*, Atlanta, Georgia, USA, February 27–March 1, 1998.
11. P. Domel, A. Lingnau, and O. Drobnik, "Mobile agent interaction in heterogeneous environment." in *Mobile Agents 97*, LNCS 1219, Springer-Verlag, Berlin, 1997, pp. 136–148.
12. R. Englemore and T. Morgan, editors, *Blackboard Systems*. Addison-Wesley, Reading, MA, 1988.
13. T. Finin, R. Fritzson, D. McKay, and R. McEntire, "KQML as an agent communication language," in *Proc. of the Third International Conference on Information and Knowledge Management*, Gaithersburg, Maryland, November 1994.
14. A. Fuggetta, G. Picco, and G. Vigna, "Understanding code mobility," *IEEE Transactions on Software Engineering*, vol. **24**(5), pp. 352–361, 1998.
15. D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. **7**(1), 1985.
16. D. Gelernter, "Multiple tuple spaces in Linda," in *Proceedings of PARLE*, LNCS 365, 1989.
17. D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. **35**(2), pp. 97–107, 1992.
18. D. Gelernter and L. Zuck, "On what Linda is: Formal description of Linda as a reactive system," in *Coordination Languages and Models*, LNCS 1282, Springer-Verlag, Berlin, 1997, pp. 187–204.
19. M. R. Genesereth and R. E. Filkes, "Knowledge interchange format: Version 3.0 reference manual," Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.
20. R. Gray, "Agent Tcl: A flexible and secure mobile-agent system," in *Proc. of the Fourth Annual Tcl/Tk Workshop*, Monterey, California, July 1996.
21. N. M. Karnik and A. R. Tripathi, "Design issues in mobile-agent programming systems," *IEEE Concurrency*, vol. **6**(3), pp. 52–61, 1998.
22. J. Kiniry and D. Zimmerman, "A hands-on look at Java mobile agents," *IEEE Internet Computing*, vol. **1**(4), pp. 21–33, 1997.
23. P. Wyckoff, S. W. McLaughry, T. J. Lehman and D. A. Ford, "T Spaces." *IBM Journal of Research and Development*, vol. 37 (3- Java Technology); pp. 454–474, 1998.
24. N. Minsky and J. Leichter, "Law-governed Linda as a coordination model," in *Object-Based Models and Languages*, LNCS 924, Springer-Verlag, Berlin, 1994, pp. 125–145.
25. A. Ohsuga, Y. Nagai, Y. Irie, M. Hattori, and S. Honiden. "PLANGENT: an approach to making mobile agents intelligents," *IEEE Internet Computing*, vol. **1**(3), pp. 50–57, 1997.
26. OMG, CORBA 2.1 specifications, 1997. `http://www.omg.org`.
27. A. Omicini, E. Denti, and A. Natali, "Agent coordination and control through logic theories," in *Topics in Artificial Intelligence*, LNAI 992, Springer-Verlag, Berlin, 1995, pp. 439–450.
28. Andrea Omicini, "On the semantics of tuple-based coordination models," in *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC '99)*, San Antonio, Texas, USA, February 28–March 2, 1999.

29. G. A. Papadopoulos and F. Arbab, "Coordination models and languages," *Advances in Computers*, vol. **46**: The Engineering of Large Systems, pp. 329–400, August 1998.
30. H. Peine and T. Stolpmann, "The architecture of the Ara platform for mobile agents," in *Mobile Agents '97*, LNCS 1219, Springer-Verlag, Berlin, 1997, pp. 50–61.
31. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," in *Mobile Object Systems*, LNCS 1222, Springer-Verlag, Berlin, 1997, pp. 49–64.