A case study in designing a document-centric coordination application over the Internet

Paolo Ciancarini and Davide Rossi and Fabio Vitali
Dipartimento di Scienze dell'Informazione
University of Bologna - Italy
E-mail: ciancarini,rossi,vitali@cs.unibo.it

Abstract

In this paper we describe and discuss a solution to the *conference management problem*, that is a case study problem in designing a groupware application distributed over the WWW. Such a problem requires supporting the coordination of network activities of people engaged in reviewing and selecting documents, namely papers submitted for a scientific conference. We discuss why such an application is interesting and describe how we designed its software architecture. The architecture we suggest implements what we call an *active Web*, because it includes agents able to use services offered by WWW infrastructures. A special kind of agents are *active documents*, which are documents that carry both some content and some code able to manipulate such a content. Users, agents, and active documents can interoperate using a set of basic services for communication and synchronisation. The active Web implementation we describe here is based on Java integrated with coordination technology.

1 Introduction

The WWW is now the most popular platform to access Internet services, so it has the potential to become the standard infrastructure to build applications integrating network services. In fact, most application domains are turning to the WWW as the software infrastructure of choice for building distributed applications leveraging on the open standards, their diffusion, and the programmable nature of the available services. For instance, there is a growing interest in document management systems based on the WWW infrastructure. These applications typically exploit *multi-agent* technologies, meaning that they are highly concurrent, distributed, and often based on mobile code written in Java [MunDew97].

However, the WWW in its current state does not provide support for document-centric applications like groupware or workflow, which require sophisticated agent coordination. In fact, most WWW applications are either server-centric (interfacing applications via CGI to a mainframe-like central server machine), client-centric (applets providing application services to users without a real distribution concept), or not integrated at all with the Web (applications whose user interface is implemented by applets or plug-ins connecting with some proprietary protocol to a proprietary server). These approaches do not really embody the idea of a programmable, *active* Web, that is based on some controllable configuration of (autonomous) agents: such approaches are either not really distributed, or not integrated with the current WWW middleware.

The PageSpace architecture [CTVRK98] provides a reference framework for overcoming these limitations, establishing some design guidelines for Web-based applications that are composed of autonomous agents performing their duties regardless of their physical positions. In this paper we demonstrate the flexibility of PageSpace by describing a case study where such an architecture has been fruitfully used.

The paper is structured as follows: In Section 2 we describe a case study in document-centric groupware over the Internet: the management of a scientific conference. In Section 3 we discuss how our idea of an active Web, that is based on PageSpace, a reference architecture for WWW-based, agent oriented interactive applications. In Section 4 we discuss how we exploited PageSpace to design a conference management system. In Section 5 we show how to extend our approach in order to include the management of active documents. Section 6 concludes the paper.

2 The case study

The purpose of this section is to informally specify a case study that concerns peer review and evaluation of submissions to scientific conferences exploiting the Internet as the infrastructure for communication and co-operation. More precisely, we intend to specify a system for supporting all the activities which typically have to be performed by a number of people widely distributed all over the

world to submit, select, and prepare the set of papers which will be published in the proceedings of a scientific conference. The goal is to build the final version of conference proceedings, including a list of accepted papers.
Following the idea that "*coordination is the management of dependencie*s" [MalCro94], conference workflow management is concerned with managing the dependencies of activities (the workflow) necessary to produce the proceedings of a scientific conference (and of course the list of authors invited to present their papers). Moreover, most of the activities to be performed are boring, long, and repetitive - e.g., fetching the papers, or filling referee report forms - and are better performed off-line.
Most scientific communities have established policies and mechanisms implementing some kind of conference management aiming at minimising the organisational efforts but keeping high the quality of papers being accepted and the fairness of the selection process. It is a usual choice that a *program committee* is established to take decisions about which papers are accepted and which are rejected. Authors interested in presenting their work submit papers to such a committee for review.
Within the committee, a group decision has to be taken according to some fair policy. The decision is usually taken by reaching a consensus or voting or ranking submissions with the help of several reviewers who help the PC members in evaluating each paper.
We believe that the proposed case study includes interesting coordination issues to study. It involves a set of coordination activities and requires coordination as a workflow itself. The case study can serve as an example of a real life application with prominent coordination aspects.
The case study is sufficiently familiar to most scientists, who normally participate to conferences as either authors, reviewers, PC members, or PC chairs. The case study is also a simple instance of a general class of problems, in which a composite document has to be produced as the result of a workflow of activities by several people.
Interestingly, this case study is inspired by some already available systems for electronic submission, collection of referee reports, and PC meeting support based on Internet communication mechanisms, namely e-mail or WWW. In fact, there currently exist some systems that use the Internet to support conference management, however they are quite different in functions, scope, and conference model supported. We do not know of any explicit definition of the requirements of a generic conference management system; the conference lifecycle we describe in this section is quite simplified and should be intended only as a specific description of a specific conference.
Ideally, PC chairs, PC members and reviewers have available reliable Internet connections and standard browsers. However, scientists usually travel a lot, thus we are interested in solutions coordinating both communication media, namely e-mail and WWW, in order to support both asynchronous and synchronous coordination.

2.1 Agents and Roles

The system we intend to specify includes several agents. We take the term *agent* as primitive; intuitively, an agent is an entity which can act autonomously; an agent can send/receive messages according to some well known protocol (non necessarily reliable): e.g. snail mail, e-mail, HTTP, or others.
All agents have unique identities; for simplicity, we define these identities as unique URLs. For instance, an agent is the conference site, named `url.of.conference`. Another agent can communicate with the conference site using different protocols:

- `http://url.of.conference`
- `mailto://url.of.conference`

Some agents are *roles*, namely they represent human users which can perform some operations. There are the following roles involved in the case study:

- author (submitter of a paper); if the paper is co-authored, the *corresponding* author is who (first) submitted the paper;
- PC chair;
- PC member;
- reviewer;
- editor of proceedings.

Each role can be represented by several agents, which differ in their URLs. For instance, a typical conference could have two PC chairs, 20 PC members, scores of reviewers, and one editor of proceedings.
The operations that each role can perform are described in the next subsection.

2.2 Dynamics

We will specify the dynamics of the workflow to be enacted defining a number of tasks which the conference management system has to support. The tasks are listed below in some arbitrary order.

1. SUBMISSION OF PAPERS. A submission form, including basic information on the submission like authors, affiliations, paper title, corresponding author, format of the paper being submitted, is available on request from the conference site (authors can ask such a form to the conference site agent). Authors submit papers to the conference site. A submission consists of the submission form appropriately filled, an ASCII file containing title and abstract, and the URL of a file containing the paper in the format declared in the submission form (e.g. in PostScript). If a submission is not conform to the above requirements, it is not accepted and its sender is invited to resubmit. If a submission is conform it is stored in the conference site and its corresponding author gets an acknowledgement.
2. BIDDING FOR PAPERS. Each PC member examines the list of submissions and selects a subset of ``interesting'' papers to review. The PC chair can alter each subset to balance the review load among PC members.
3. DISTRIBUTION OF THE PAPERS TO THE REFEREES. After the deadline for submissions is expired, an agent sends by e-mail another agent to each PC member which, when run on the PC member's local system, fetches from a given URL all the papers assigned to that member for refereeing, and on request generates the forms for reviews to be sent by e-mail. Unauthorised access to all submitted papers has to be prevented, so that the list of the submissions itself -- not to mention the papers -- remains confidential.
4. COLLECTION OF THE REPORTS. A WWW form is provided for online input of referee reports into the conference site. Also, forms can be obtained by sending an e-mail message to the conference site, and forms filled off-line can be submitted directly by e-mail. In this and in the following phases there are some security and authentication issues at stake. The confidentiality of information has ensured by using passwords to allow access only to PC members, and by using a unique address for each PC member to which all communication is directed. This phase is managed by an agent that answers to requests of forms and collects the incoming reports. The agent generates a file for each submitted paper in which it collects the relative reports.
5. PREPARATION OF STATISTICS. An agent scans all paper reports and fills up a table in a standard spreadsheet (e.g. MS EXCEL) to generate statistics and an initial proposal for ranking all papers to be used during the PC meeting.
6. PC MEETING. During this phase a PC member needs to be able to:
   - access the list, the abstracts, and the files of the submissions;
   - access the referee reports collected so far for specific papers;
   - access the logs of comments on specific papers;
   - access the ranking table of all submissions;
   - read or write comments about a paper on a log file reserved for such a task;
   - finally, decide on acceptance status and record such a decision.

   All support is provided via links from a WWW page (password protected) to appropriate agents. In addition, all the features are available by e-mail, by sending messages with suitable subjects to the mail server agent. The situation gets more complex if PC members are allowed to submit papers. In this case one must deny them information about their own submissions, and, possibly, also the possibility of inferring such information.
7. COMMUNICATION OF RESULTS. Once the list of accepted papers is formed, an agent takes care of communicating the results and the referee reports to the authors. Other agents generate a list of the abstracts of the accepted papers (in HTML format), extract a list of subreferees for the referee reports, prepare a synopsis of the result for the PC Chair.
8. SUBMITTING CAMERA READY. Authors of accepted papers submit camera ready versions. An agent controls compliance with camera ready standards and warns late authors, if any.

9. PREPARING THE PROCEEDINGS. An agent collects all camera ready versions, asks and waits for a preface from the editor of proceedings, and finally prepares a draft of the proceedings. The editor and the PC chairs are informed by e-mail when the proceedings are ready.

2.3 Motivation

The case study outlined above has been inspired by the personal experience of the authors of [CNT98] as PC members and by ideas described by designers of some conference management systems in [Sas96,MatJac96,Nie97,Tol98b]. In fact, there are already several Web-based conference management systems which are being used for supporting peer review. In our experience the most robust and rich in features are the systems supporting the WWW and the AAAI conferences. Information on these systems is scarce; our description of informal requirements above is an attempt to summarise the functions offered by some existing systems. Interestingly, all the systems we have examined support quite different conference organisational models. For instance, there are systems supporting conferences organised as a set of workshops; there are systems were papers are classified by authors according to some keyword systems, and then they are automatically assigned to PC members after they have selected a set of keywords representing their "reviewing ability".
Most interestingly, only the simplest systems assume a centralised repository of papers and reviews. The system used for the WWW conference instead allows the authors to submit only the URL of their paper, that is then directly accessed by reviewers only when necessary.

3 Designing an active Web

The WWW was born as a hypertext document browsing system. The components of such a system are servers, clients, and documents. The interactions among these components are driven by the HTTP protocol and the CGI-BIN mechanism. HTTP is a very simple TCP/IP based protocol used by clients in order to retrieve documents stored on a disk under the control of an HTTP server. Usually documents are in HTML format, a mark-up language used by browsers to display the content. However, documents can also be created on-the-fly as it happens with CGI-BIN. With CGI-BIN a client requests a document as the output of a process that is run by the server when the request comes in. Things are really as easy as that but this simple mechanism can be used in a "tricky" way in order to implement more complex forms of interaction among documents, clients, and servers.
According to its original design, the only activity that can be dynamically triggered in the WWW is associated to the CGI-BIN mechanism. Soon users demanded more interaction than just browsing documents and this brought to the development of a family of languages that can be embedded into an HTML document and executed by the user's browser. These languages (or even architectures to transfer executable code, as in the case of Microsoft's ActiveX) are very different in capabilities and target; some in fact are scripting languages intended to interact heavily with the document itself (as in the case of JavaScript), some are complex and full-fledged languages that have little interaction with the document (as in the case of Java).
Nevertheless, these technologies give us the ability to "activate" two key components of the WWW architecture, namely servers and clients. However, we still lack techniques and protocols to allow these components to interoperate. Usually, in fact, current projects aiming at the exploitation of the WWW as an active distributed platform locate computing components just on one side (either at the server or at the clients). Our approach is different because we redefine the coordination capabilities of WWW middleware in which the activity takes place.

3.1 The WWW as an agent world

Here we are interested in "solving" the case study described in Sect. 2 designing an WWW-based, agent-oriented software architecture, namely what we call an *active Web*. Put very simply, an *active Web* is a system including some notion of programmable agents performing some coordinable activities exploiting network services. The activities are coordinable, menaning that they can take place concurrently at the client, at the server, at the middleware level, at the gateway with another active software system (e.g. an external database, decision support system or expert system) or even at the user level. An active Web includes several *agents*, with a well-defined behaviour. We actually see each component of an active Web as an autonomous agent in a world offering coordination services to agents: thus, an agent is not just capable of computations but it also should be able to interact (in possibly complex ways) with other agents.
The interaction among agents is usually accomplished using client/server architectures (as in any RPC-based system, such as CORBA). However, sometimes the client-server framework misses its

main goals (namely, modular design and simple interaction behaviour) when the interactions among the components change in time and the client/server relationship can be reversed, or when the designer needs decoupling among components (that is something we really need when we have to deal with a world-wide distributed system including heterogeneous networks).
A solution to such a problem consists in designing the distributed application as an agent world in which agents are spatially scattered and act autonomously: such a schema fits quite well into the distributed objects model. In our coordination-based approach agents perform sequences of actions which are either method invocations or message deliveries. Synchronisation actions (e.g. starting, blocking, unblocking, and terminating an activity) are the remaining mechanisms in object invocation. More precisely, we distinguish between *agent computation*, that is what concerns the internal behaviour of an agent, and *agent coordination*, that is what concerns the relationship between an agent and its environment, namely synchronisation, communication, and service provision and usage.
Coordination models separate coordination from computation, not as independent or dual concepts, but as orthogonal ones: they are two dimensions both necessary to design agent worlds. A coordination language should thus combine two languages: one for coordination (the *inter-agent actions*) and one for computation (the *intra-agent actions*). The most famous example of a coordination model is the Tuple Space as in Linda, which has been implemented on several hardware architectures combining it with different programming languages.
Linda can be seen as a sort of assembly coordination language in two ways. First and foremost, it offers very simple coordinables, namely active and passive tuples, which represent agents and messages, respectively; it offers a unique coordination medium, namely the Tuple Space, in which all tuples reside; it offers a small number of coordination primitives. Second, Linda is a sort of coordination assembly because it can be used to implement higher level coordination languages. For instance, we have used it to implement Jada, a library which adds coordination primitives to Java [CiaRos97].

3.2 The PageSpace

PageSpace is a reference architecture for multi-agent applications built on top of the WWW [CTVRK98]. PageSpace applications consist of a number of distributed agents to be coordinated to accomplish some cooperative task.

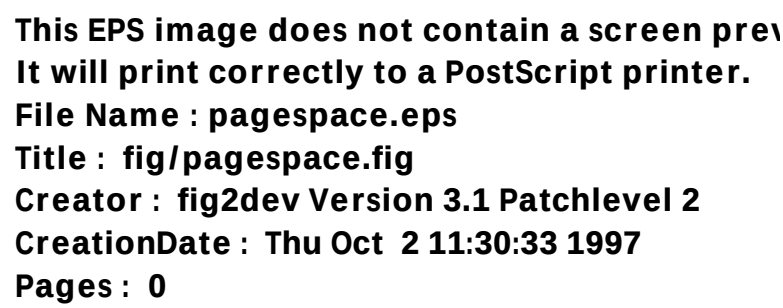

Figure 1. The PageSpace reference architecture

PageSpace includes a number of agent types, depicted in Fig.1.

- Alpha or *user-agents*, are user interface agents. Alpha agents consists of applets, scripts, and HTML pages.
- Beta or *avatars*, are a persistent representation of users on the PageSpace. An avatar allows a user to access the application agents on the PageSpace, provide the user with the required Alpha agents, and collects the messages addressed to an Alpha agent or the user himself in his absence.
- Delta or *application agents*, perform the computations of the applications. They offer services by interacting with the shared data. Some delta have to interact with the user, and therefore must be able to produce an interface, usually in the form of an alpha agent that is then downloaded to the client. Others do not directly interact with the user, and just offer services to other application agents.

· Zeta or *gateway agents*, allow a PageSpace to interact with external environments such as other coordination environments or distributed applications residing outside the PageSpace.

Every browser includes at least one user-agent, which is connected and interacts directly with an avatar, running on a PageSpace server. A set of Delta agents implement the coordination mechanisms necessary to an active Web application. Gateway agents provide access to external services, like e-mail or a CORBA ORB.
Another key concept in PageSpace is the *coordination kernel* (Gamma) which is used for the specific implementation of an application. Several coordination kernels exist, based on different coordination models and offering different services; PageSpace is independent from a specific one. In this paper we will use Jada [CiaRos97].

3.3 Jada coordination mechanisms for agents

In [CiaRos97] we introduced Jada, a Java-Linda combination. Jada was a first experiment in combining Java with a coordination model, and provides some basic coordination mechanisms. Here we describe an enhanced version, namely MJada, which adds to Jada the support for coordination among mobile agents.
A coordinable agent in MJada is a valid Java program which can access a coordination medium. The MJada coordination medium is the tuple space, like in Linda [CarGel92]. However, Linda uses only one tuple space, although physically distributed. Instead, MJada supports *multiple tuple spaces*, which are physically distributed over the Internet and logically nested and thus form a hierarchical coordination structure based on the `TupleSpace` object. Hence, a `TupleSpace` offers some `navigation" methods which allow a thread or an agent to navigate the coordination structure, expressing “localities” and “itineraries” as sequences of names similar to UNIX file system names. Tuple space names are expressed by paths, so both the strings "`/space`" and "`/chapter/section`" are valid names.
A `TupleSpace` object can be created with the Java construct:

```
TupleSpace tuple_space = new TupleSpace();
```

An agent can connect a tuple space with the `join()` method. For instance, after the agent executes the following statement

```
tuple_space.join("space1");
```

all operations it performs will be relative to the tuple space called `space1`.
Tuple space names can be specified as either relative or absolute paths. For instance, if the current tuple space is "`/working_space`" and `join` is called with argument "`/games_space`", the subsequent operations are performed on space "`/games_space`". It is also possible moving to the encompassing space specifying ".." as argument of `join()`. For flexibility additional methods are provided a) to move to the encompassing tuple space, `leave()`, or b) to the root tuple space, `leaveAll()`.
A tuple is represented by the `Tuple` class and contains a set of Java objects. We can create a tuple as follows:

```
Tuple tuple = new Tuple("Hello!", new Integer(1));
```

Tuples can be inserted in a tuple space with the `out()` method.

```
tuple_space.out(tuple);
```

Tuples are retrieved from a tuple space using an associative mechanism: when an agent calls the `In` method it has to pass it a tuple that is used as a matching pattern. The `In` method returns a tuple (if any) that matches the given pattern (the same applies to the `Read` method; the difference between `Read` and `In` is that `In` removes the returned tuple from the tuple space). In order to have flexible matching operations we introduce the concepts of *formal* and *actual* tuple items: a *formal* item is an instance of the `Class` class (the meta-class used in Java). Any other object is an *actual* item.

Two tuples match if they include the same number of items and each item of the first tuple matches the corresponding item of the second tuple. For instance, the following is a template tuple with an actual field of class `String` and a formal field of class `Class`.

```
Tuple template=new Tuple("Hello!",new Integer(0).getClass());
```

The tuple `template` matches both `t1` and `t2` created as follows:

```
Tuple t1 = new Tuple("Hello!", new Integer(3));
Tuple t2 = new Tuple("Hello!", new Integer(7));
```

Tuples can be read or withdrawn from a space using the following methods

```
- Result read(Tuple formal)
- Result in(Tuple formal)
```

Differently from Linda, disruptive MJada operations do not return a result tuple, but a place-holder for that tuple represented by the `Result` class. The place-holder can then be used to test result availability, to fetch a result or to kill the request. Trying to fetch a tuple that is not available will block the calling thread, resulting in the same coordination mechanism used in Linda.
Tuple spaces in MJada can either be "local", namely shared among concurrent threads running in the same Java Virtual Machine, or "remote", namely running on a (possibly) remote host and accessed via an ad-hoc proxy class in a way that is similar to the one used by RMI.
The main feature of MJada to support mobile agents coordination is the ability of transparently abort and re-send a request for a pending `In` or `Read` operation among migrations. Thus, if an agent performs:

```
Result result = remote_tuple_space.In(my_tuple);
```

and the requested tuple is not available at call time, it can migrate to another place and the `result` object will still refer to a valid `In` operation performed on the remote tuple space. MJada provides also multiple-result operations that allow one to read or withdrawn all tuples that match a given template.

```
- Enumeration readAll(Tuple formal)
- Enumeration inAll(Tuple formal)
```

Result tuples can then be fetched using Java enumeration constructs; Linda does not have any similar operation. In addition to the previous basic tuple operations Jada introduces a new coordination mechanism based on tuple *collections*. A tuple collection, represented by the `TupleCollection` class, defines a sequence of tuples having the same signature. In order to build a tuple collection we write

```
TupleSpace space = new TupleSpace();
Tuple pattern = new Tuple(new String().getClass(),
                          new Integer.getClass());
TupleCollection tc = new TupleCollection(space,pattern);
```

where space is the tuple space where collected tuples reside and pattern is a tuple which define the tuples' signature. Tuples can be inserted in a collection using the `add()` method

```
tc.add(new Tuple("Hello!", new Integer(1)));
```

If the specified tuple has a signature different from the collected one, an exception is thrown. The main feature of collections is that tuples can be read or withdrawn in the same order they were inserted with *iterator* objects. Two predefined iterators are provided, `ReadIterator` and `InIterator`, but more advanced iterators can be added to the framework, provided that they implements the `nextTuple()` method. For instance, the following code reads all tuples from the previous collection

```
TupleIterator iterator = tc.readIterator();
Tuple result;

while( ... ) {
  // get next tuple in this collection
  result = iterator.nextTuple();
  // use tuple items
  ...
}
```

Tuple collections and iterators capture a recurrent pattern of coordination programming, that is consuming a sequence of tuples, and simplify noticeably the source code. Iterators are not built using a constructor, but with a *factory* method of the `TupleCollection` class. The main advantage is the possibility to develop extended collection classes which use the same factory method to create iterators.
Iterators are also a well known *design pattern* [GHJV95] used to encapsulate access and traversal logic of an object container. Thus different iterators implements different access policies, leading to an improved design of the system. The main feature of `ReadIterator` and `InIterator` is that the `nextTuple()` method is blocking. This way we can define an iterator that reads all tuples already inserted in the collection, but also all tuples that will be inserted in the future. The same result is obtained in other coordination languages like for instance Linda using an index as a tuple field; iterators offer a more elegant solution.

3.4 Designing an application using PageSpace

The design of multi-agent, interactive applications over the WWW is driven by PageSpace as follows. The designer has to define a topology of tuple spaces to be mapped onto some tuple space servers written with MJada. Then Java programs can be created which access the tuple spaces: all tuple space operations should specify a pathname; if no pathname is used, the operation refers to the default reference tuple space. PageSpace offers a library of classes which implement the basic agent types (e.g. alpha, beta, etc.).
In the next section we describe how MJada has been used to implement an instance of PageSpace called MUDWeb.

4 Designing a conference management system

PageSpace is a reference architecture, independent from a specific coordination kernel or language. MUDWeb is an actual software architecture we designed inspired by PageSpace. MUDWeb takes its name from MUD (Multi User Dungeon), a cooperative interactive environment shared by several people to socialise and interact.

4.1 MUDs as platforms for groupware

MUDs have been proposed as enabling technologies for some kinds of groupware applications [Das97,DHW98]. A MUD usually represents the communication infrastructure of a role-playing game (hence the name) where human and robot players interact, visit dark and magical places, fight monsters or other players, and seek treasures. More generally, a MUD is a programmable platform that creates shared virtual realities accessible over a network. Thus, a MUD is a very powerful abstraction to describe a general platform for cooperative work, that provides a general framework for users to interact with each other, and with resources such as documents.
Any MUD, generally, is based on the concepts of *rooms*, *items* and *players* (or *users*). The whole virtual space inside a MUD is partitioned in rooms. Each room can contain several users and items. The rooms are connected in various ways. Each user can move from a room to another one can interact only with the items in its same room: a user can take an item, use it, and so on. From this point of view items are passive entities that are “activated” by the users. Moreover, interactions among users can take place only if they are in the same room (note that in this context the world room not necessarily means a closed place: a room in a MUD might virtually represent a cave in a dungeon or a garden around an enchanted castle, i.e. it is a partition of the virtual space in which interactions take place).

As we pointed out wandering inside a MUD a user might meet robot players. As the name suggests a robot player behaves just like other users but there is not its human counterpart: the actions of the robot player are driven by programs. We can think of a robot player as a synthetic user or as an active item; we shall see later that these two metaphors can lead to very different implementations.

4.2 MUDWeb: implementing a MUD with MJada

Implementing a MUD using a system based on multiple tuple spaces like MJada is quite straightforward: we can use a tuple space for each room; an item is a tuple stored in the tuple space that represents the room that contains the item each user is an agent. Robot players are synthetic users: programs that access the MUD using the MJada primitives. We can use robot players to provide simple services to other users (in fact we will often refer to robot player as to server agents). By interacting with a server agent, the users can activate a service and, eventually, gather its output. Since the relationship among agents in MJada takes place with tuples exchange this same protocol is used to activate the services of the server agents.
The software architecture of MUDWeb consists of a number of services which agents can use according to a number of protocols based on tuple exchanges. MUDWeb includes several rooms which correspond to tuple spaces including some basic server agent. The whole architecture is implemented by a server organised as nested tuple spaces. Figure 2 shows the software architecture of MUDWeb.

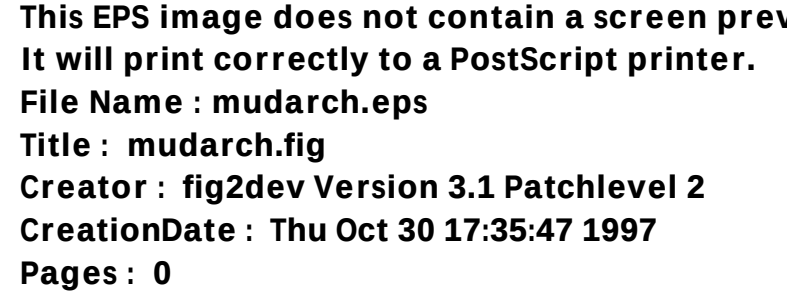

Figure 2 A MUD-like active Web

Services wait for command tuples and perform services based on their content. Services are generally very simple and specialised agents that react to a limited list of commands. The functionalities of an application are thus implemented by a score of services cooperating together. Clients are user interfaces using some role-specific HTML page or Java applet. We have used the enhanced Jada as the coordination environment, used by all agents to coordinate their activities.
An agent server supports three kinds of agents: the *avatars*, the *services*, and the *mudshell*. An avatar is the persistent representation of a human user. The avatar dialogues with the user interface to be displayed within a WWW browser, and can accept commands and return data in a variety of methods, including e-mail messages. Services are the Delta agents, i.e. the modules on the shared space that provide the actual computations of the distributed application. The Mudshell is the client of the MUDWeb application, and is the interface framework where the interaction with the user takes place: the MUDshell provides primitives for moving from one available shared space to another, and allow the user to interact with the services by providing a MUD-like text box for direct commands, and displaying the most common ones on appropriate buttons. Furthermore, they allow avatars to display their interfaces just like the other services.

4.3 A conference management system designed as a MUD

MUDWeb has been used to design and implement a workflow system supporting the management of a scientific conference. The system has the goal to simplify the management of the review of papers submitted to a scientific conference.
Submitted papers are stored in rooms; authors, reviewers, and program committee members are represented by avatars which support both synchronous (online) and asynchronous (e-mail) communication interactions.

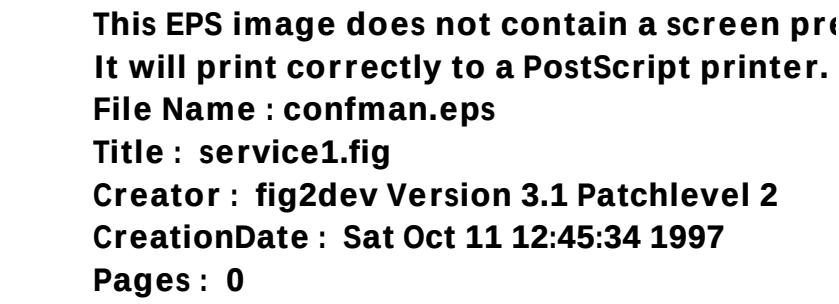

Figure 3 Conference management mapped onto MudWeb

The system architecture is depicted in Figure 3. It includes the following rooms:

- SubmittedPaper. Every paper is stored in a room of this type, that is dynamically created when the paper is submitted. The room will also store the reviews when they will be ready.
- ReviewRoom is used by reviewers to store their reviews before they are finalized.
- SelectRoom is a room accessible to the program committee members only; it stores the scores assigned to papers.
- Papers is a room reserved to the conference organizers: it contains managing data like the full list of submitted papers and the address data of authors.

MUDWeb offers in each room some specific services:

- Services for authors of papers. When a paper arrives it is stored in a room created on purpose, which includes also an avatar representing the corresponding author. The avatar can answer simple questions on the status of the submission.
- Services for reviewers. Each *SubmittedPaper* room, created to store a submitted paper, can also store reviews for such a paper. User input is handled by HTML forms and two CGI scripts. The CGI scripts generate a form and build a tuple, respectively. The tuple is temporarily stored in the same room containing the paper being reviewed until it is “confirmed”; after the confirmation the tuple is moved in the selectRoom.
- Services for PC members. The service *Services.Selector* can be activated in the room *SelectRoom* and is implemented by the class `Services.Selector`.

All services come in two flavours, synchronous and asynchronous. For instance, the service *Services.Announcer* accepts reviews coming by e-mail, checking that they refer to the paper stored in the same room. The service *Services.Submitter* is similar to the preceding one, but supports on-line user interaction. Asynchronous services rely upon *avatars*, which have to be programmed to perform the necessary tasks. The (server-side) avatar acts as an e-mail client of an e-mail server; it controls the user mailbox and processes the messages it contains.

5 Active documents: enters XML

As we pointed out above, MUD's robot players can be seen as either artificial users or as active items. As of now we pursued only the first approach. It is however evident that there are classes of applications in which the documents we have to deal with can be “active”, i.e.they are subject to auto-modifications.
An active document that represents a stock exchange's stock, for example, should periodically update its own value. An active document that represents a contract should change its own state (and maybe also warn some user) when its expiration date is reached. In a MUD framework the mapping of active documents into artificial players is not natural and often not even correct: players and active documents are different concepts. It seems evident that active documents should be mapped into programmable entities. The problem we face, using this approach, is that we need a standard framework that enable us to represent both the contents of the document and its semantic.
We expect that multiagent, document-centric applications will exploit the forthcoming XML technology, which allows to define *active documents*, namely messages which carry not only contents but also code able to manipulate such a content. XML [BPS97] is an extensible document markup

language that provides a unified framework for describing orthogonally a document structure, its rendering, and its semantics. Introducing a technique that we call "*displets*" [CVM99], our group has integrated XML with Java, thus we can now use a Turing-equivalent language for activating an XML document. The purpose of displets is to create active documents that are able to render themselves. We are planning to re-use the same concepts that are at the base of the displets to implement some more general-purpose kind of active documents.
The integration of this kind of active documents into the active items of a MUD environment based on MJada is very easy and opens new and interesting opportunities to design an active document management system based on the MUD metaphor, the PageSpace architecture and the Mjada library. We are currently engaged on this work.

6 Conclusions

We have presented a case study in document-centric groupware. We know that several systems exist which support conference management; some of them work over the WWW or over proprietary platforms like Lotus Notes. We have sketched a solution using PageSpace, an agent-based reference architecture used to design an actual software architecture based on the MUD metaphore.
The case study we have exposed is intended as a benchmark to compare modern object oriented middleware infrastructures: for instance, we are developing a similar conference management system based on Lotus Notes and in a future paper we intend to compare it with the present solution based on PageSpace. Another issue we are exploring is security, that is especially important for document-centric multiuser applications.
We have already noted that an interesting feature of the case study is that there exist several "conference models", and that a truly flexible system should be able to support all of them. An issue that we have not discussed in this paper is what happens if documents to be managed are *active*, ie. they include not only some contents but also some code. In fact, possibly the most interesting issue we are exploring is the integration in a coordination environment of active documents written in XML integrated with Java. We expect that coordination technology offers further degrees of integration and flexibility.

Acknowledgments.
PageSpace has been supported by the EU as ESPRIT Open LTR project 20179 as a joint research with people in TU Berlin. The case study requirements have been defined together with R.Tolksdorf (TU Berlin, Germany) and O.Nierstrasz (Univ. of Berne, Switzerland), with partial support from EU WG ``Coordina" project. Our students R.Gaggi and A.Giovannini helped with the implementation of the conference management system. More information and the code are available at the site `www.cs.unibo.it/~rossi`.

References

[**BPS97**] T.Bray, J.Paoli, and C.Sperberg-McQueen. Extensible Markup Language (XML). *The World Wide Web Journal*, 2(4), 1997.

[**CarGel92**] N.Carriero and D.Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97-107, 1992.

[**CNT98**] P.Ciancarini, O.Niestrasz, and R.Tolksdorf. A case study in coordination: Conference Management on the Internet.

`ftp://cs.unibo.it/pub/cianca/coordina.ps.gz`, 1998

[**CiaRos97**] P.Ciancarini and D.Rossi. Jada: Coordination and Communication for Java agents. In J.Vitek and C.Tschudin (eds), *Mobile Object Systems: Towards the Programmable Internet, LNCS* 1222:213-228. Springer-Verlag, Berlin, 1997.

[**CTVRK98**] 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*, 24(5):362--375, 1998.

[**CVM99**] P.Ciancarini, F.Vitali, and C.Mascolo. Managing complex documents over the WWW: a case study for XML. *IEEE Transactions on Knowledge and Data Engineering*, (to appear), 1999.

[**Das97**] T.Das et al. Developing Social Virtual Worlds using NetEffect. Proc. 6th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 148--154, Boston, June 1997. IEEE Computer Society Press.

[**DHW98**] J.Doppke, D.Heimbigner, and A.Wolf. Software Process Modeling and Execution within Virtual Environments. ACM Transactions on Software Engineering and Methodology, 7(1):1--40, January 1998.

[**GHJV95**] E.Gamma, R.Helm, R.Johnson, and J.Vlissides. *Design Patterns*, Addison-Wesley, 1995.

[**MalCro94**] T.Malone and K.Crowstone. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87-119, 1994.

[**MunDew97**] J.Munson and P.Dewan. Sync: a Java Framework for Mobile Collaborative Applications. *IEEE Computer*, 30(6):59-66, 1997.

[**MatJac96**] G.Mathews and B.Jacobs. Electronic Management of the Peer Review Process. *Computer Networks and ISDN Systems*, 28(7-11):1523, Nov. 1996.

[**Nie97**] O.Niestrasz. Identify the champion. `www.iam.unibe.ch/oscar/PDF/champion.fm.ps`, 1997.

[**Sas96**] V.Sassone. Management of electronic submission, refereeing, and PC meeting. (Manual of a WWW system), Nov. 1996.

[**Tol98**] R.Tolksdorf. Conference reviewing. `grunge.cs.tu-berlin.de/tolk/reviewing.html`, 1998.