The World Wide Web as a Place for Agents

P. Ciancarini[1] and Robert Tolksdorf[2] and F. Vitali[1]

[1] Dipartimento di Scienze dell'Informazione
Mura Anteo Zamboni, 7 – 40127 Bologna, Italy
University of Bologna - Italy
E-mail: {ciancarini,vitali}@cs.unibo.it
[2] Technische Universität Berlin, Fachbereich Informatik, FLP/KIT,
Sekr. FR 6–10, Franklinstr. 28/29, D-10587 Berlin, Germany,
mailto:tolk@cs.tu-berlin.de, *http://www.cs.tu-berlin.de/~tolk/*

Abstract. The Word Wide Web was born as an Internet service supporting a simple distributed hypertext management system. Since its start a number of technologies have been proposed to enhance its capabilities. In this paper we describe our concept of an *active Web*, namely how we design the software architecture of interactive cooperative applications based on the Word Wide Web. An active Web includes *agents* able to use the services offered by Word Wide Web clients and servers. In an active Web both users and agents can interoperate using a set of basic mechanisms for communication and synchronization. The active Web we describe here is based on coordination technology: we explore two alternative implementations, both based on Java enriched with alternative coordination kernels.

1 Introduction

The Word Wide Web is currently the de facto standard platform to access several Internet services, like FTP or USENET news. Given its pervasivity, it has the potential to become the most used infrastructure to build geographically distributed applications as well. In fact, several enterprises are turning to the Word Wide Web as the environment of choice for building innovative and sophisticated applications leveraging on the open standards, their diffusion, and the programmable nature of the environment. In fact, the Internet can be used as an effective and economical infrastructure for business support. For instance, there is a growing interest in groupware applications based on the WWW infrastructure. These applications are typically *multiagent*, meaning that they are highly concurrent, distributed, and based on mobile code (Munson und Dewan, 1997).

However, the Web in its current state does not provide enough support for applications based on agent-oriented programming, like groupware or workflow, which require multi-agent coordination and complex transactions. Most WWW applications are either server-centric (applications interfacing via CGI to a central mainframe-like server machine), client-centric (applications made mostly of applets providing services to users), or not integrated at all with the Web (applications whose user interface is implemented by applets or plug-ins using some proprietary protocol to connect to a proprietary server). All these approaches do not really satisfy the idea of an active Web based on some structured configuration of (autonomous) agents: they are either not really distributed, or not integrated with the Web.

The PageSpace architecture (Ciancarini et al., 1998) provides a design framework for overcoming these limitations, creating a new paradigm for Web-based applications that are composed of autonomous agents performing their duties regardless of their physical locations. The PageSpace architecture is based on the Linda coordination model (Carriero und Gelernter, 1992) and can be controlled with different coordination languages. In this paper we demonstrate the flexibility of the architecture by describing two different coordination languages and applications where the PageSpace architecture has been fruitfully used.

The paper is structured as follows: in section 2 we discuss the issues arising when one has to design distributed, interactive applications on the WWW. In section 3 we illustrate the PageSpace reference architecture, that can be used to design these applications. In section 3 we discuss a direct implementation in Java of the PageSpace reference architecture. In section 5 we discuss how we implemented a PageSpace-based application using Java enriched with some coordination primitives. In section 6 we discuss alternative research and commercial efforts to provide interactivity and coordination on the Web. Section 7 concludes the paper.

2 Designing active Web applications

The Word Wide Web was born as a simple and efficient distributed system for managing hypertext documents. The minimal set of components of such a system are servers, clients, and documents including hyperlinks. The interactions among these components are driven by the HTTP protocol and the CGI (Common Gateway Interface) 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 stored in HTML format, a mark-up language with a typographical semantics used by browsers to display the document contents. However, documents can also be created on-the-fly by a server-side application. With CGI a client requests a document as the output of a process that is run on the server when the request comes in. This simple mechanism can be used in a "tricky" way in order to implement more complex forms of interaction among documents, clients, and servers (Bannan, 1997).

According to the original design the only activity that can be dynamically triggered in the Word Wide Web is associated to the CGI mechanism. Soon users asked for 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 are very different in their capabilities and target; in fact, some are scripting languages intended to interact heavily with the document itself (as in the case of JavaScript or ActiveX), some are complex and fully-fledged languages that have minimal interaction with the document (as in the case of Java).

Although these technologies allow us to "activate" the two key components of the WWW architecture, namely servers and clients, we still need techniques and protocols to control the coordination of such components.

2.1 The WWW as an agent world

Intuitively, an *active Web* is a system based on standard WWW clients and servers including some kind of agents performing some activities on Internet resources. The activities can take place at the client, at the server, at the middleware level, at the gateway with another active software system (eg. an external database, a decision support system or an expert system), or even at the user level.

An active Web includes several *agents*. In fact, each component of an active Web is as an autonomous agent in a world offering several services to agents: thus, an agent is not just capable of computations but can also interact (in possibly complex ways) with other agents.

The interaction among agents is usually accomplished using a client/server architecture (as in any RPC-based system, such as CORBA) (Lewis, 1995). However, sometimes the client-server architecture is not adequate: the interactions among the components may change in time and their client/server relationship needs to be reversed, or a designer may need to decouple the components, for instance to deal with an application distributed over a world-wide intranet including heterogeneous networks and machines.

A solution to these problems consists of designing the distributed application as made of spatially scattered agents acting autonomously: such a schema fits quite well into the so called *distributed objects model* (Adler, 1995, Orfali et al., 1996). When an application is composed of several agents, the control of their coordination is an important issue. In our approach we distinguish between agent computation and agent coordination.

Agent computation is what concerns the internal behavior of an agent. An agent performs actions which are either method invocations or message deliveries. Synchronization actions (eg. starting, blocking, unblocking, and terminating an activity) are the remaining mechanisms in object invocation.

Agent coordination is what concerns the relationship between an agent and its environment, namely synchronization, communication, and service provision and usage.

Coordination languages separate coordination from computation, not as independent or dual concepts, but as orthogonal ones: they are two dimensions both necessary to design agent worlds (Ciancarini, 1996).

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 Linda's Tuple Space

(Carriero und Gelernter, 1992, implemented on several hardware architectures and integrated with several programming languages.

Linda can be seen as a sort of assembly coordination language in two ways. First and foremost, it includes very simple coordination entities, active and passive tuples, which represent agents and messages, respectively; it includes a generic coordination medium, namely the Tuple Space, in which all tuples reside; it includes a small number of coordination primitives. Second, we say that Linda is a minimalist coordination assembly because it can be used to implement higher leve,l more complex coordination languages. For instance, we have used it to implement Shade, an extension of Java including a concept of transactions (Ciancarini und Rossi, 1998).

In fact, the Linda coordination model is quite minimalist to be immediately used to coordinate WWW components (Bjornson et al., 1997). What lacks is the definition of a *reference architecture*, including the basics of tuple space coordination but adding specific services for WWW agents. The interest for coordination architectures is rapidly rising in the industrial world: CORBA can be considered one of these architectures. Another one is Jini, by SunSoft. In the next section we describe PageSpace, that is an example of such an architecture.

3 The PageSpace

PageSpace is a reference architecture for multiagent applications built on top of the WWW (Ciancarini et al., 1998). PageSpace applications consist of several distributed agents which coordinate themselves to accomplish some cooperative task.

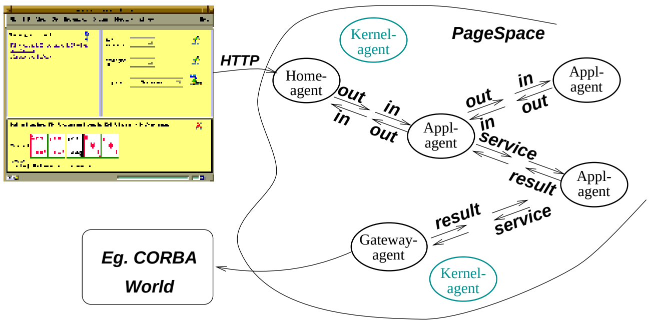

Fig. 1. The PageSpace reference architecture

In the PageSpace reference architecture, therefore, we distinguish several types of *agents*, as depicted in Fig. 1:

– *User interface agents* (also called *alpha agents*), implement application-specific user interfaces in general-purpose browsers. They are displayed in a browser according to the requests of the user. Depending on the complexity of the application and the capabilities of the browser, there may be different instantiations of user interface agents (in HTML, JavaScript, Java, etc.) that are displayed or executed on the browser.

– *Homeagents* (also called *beta agents*) are a persistent representation (avatar) of users in a PageSpace. Even if at any moment users can disconnect from the shared workspace, it is still necessary to collect, deliver, and possibly act on the messages and requests of the other agents. The homeagent receives all the messages bound to a user, and delivers them orderly to the user on request (ie. when the user is on-line). Evoluted homeagents can in some circumstances actively perform actions or provide answers on behalf of the user in her absence.
– *The coordination kernel* (or *gamma*) is the actual operating environment where agents live and communicate. In PageSpace we do not dictate the use of a specific coordination kernel. In fact, different coordination kernels may provide different mechanisms for creating and controlling the agents of an application. Thus, a designer should choose the coordination kernel most suitable to his application. In this paper we will describe two different coordination kernels we have used, both based on Java.
– *Application agents* (also called *delta agents*), are the agents that actually perform the working of the coordinated application. They are specific of one application, and can be started and interrupted according to the needs of the application. They live and communicate in the coordination kernel, offer and use each other's services, interact with the shared data, and realize useful computations within the PageSpace.
  Some application agents will not interact directly with a user in any way, and therefore will have no need for a user interface. They will just use or offer services to other agents. Other application agents, on the other hand, will have to be controlled and monitored by a user. In this case, they will provide on request an ad hoc user interface agent, in the form of an HTML document, a Java applet, etc., which will be delivered to the requesting user and will be displayed or executed on the user's machine.
– *Kernel agents* (also called *epsilon agents*), provide system services to the application agents. They perform management and control task on the agents active within the PageSpace environment. They deal with the activation, interruption, and mobility of the agents within the physical configuration of connected nodes. Ideally, there would be one kernel agent for each participating machine, providing access to the local workspace. Kernels maintain the illusion of a single shared PageSpace when it is actually distributed on several computers, and provide mobility of the agents on the different machines for load balancing and application grouping as needed.
– *Gateway agents* (also called *zeta agents)*, provide access to the external world for PageSpace applications. Applications needing to access other coordination kernels, network services, legacy applications, middleware platforms, etc., may do so by requesting services to the appropriate gateway agent. Each gateway agent is specialized for dealing with one type of external en-

vironment, and will translate and deliver externally the services requests of the application agents, and will deliver the corresponding response back to the appropriate agent.

We call "agents" the entities present in the PageSpace architecture since they are more complex than pure objects: the application agents are autonomous and possibly mobile, homeagents can include knowledge bases to work on behalf of a user, etc.

We foresee a user interface agent in each user browser, which is connected to a homeagent providing stable access to the PageSpace. A set of application agents implement the functionality of a distributed application, and a gateway agent provides access to a external environment, for instance, some OLE-based document-centered applications, like a spreadsheet, or some network service, like e-mail or news.

4 A prototypical implementation of PageSpace

In this section we describe one initial instance of the PageSpace reference architecture that has been implemented in Java for evaluation purposes. Since this prototypical implementation never had any other name, we will inappropriately use here the term "PageSpace" to refer to it, rather than to the reference architecture described in Ciancarini et al., 1998, as we do in the rest of the paper.

The user interacts with his homeagent via the user interface agent as shown in a Web browser. The homeagent is identified by a normal URL, such as `http://pagespace.cs.tu-berlin.de/cgi-bin/beta`. A specific homeagent is identified by the URL plus a user name, such as `http://pagespace.cs.tu-berlin.de/cgi-bin/beta/robert`.

After logging into the homeagent, the browser displays an user interface such as the one shown in Figure 2. The list on the top left contains all the messages received from all the application agents needing to communicate with the user. The control area on the top right offers the user a set of buttons to control the homeagent. Finally, by clicking on one of the messages in the list an appropriate application-specific user interface is shown in the display area on the bottom.

Specifically, the steps involved in using an application are the following ones:

1. The user selects an application from a list in the control area and instructs the homeagent to use it.
2. The homeagent retrieves additional data on the application via HTTP and the initial interface for that application (a static HTML document stored somewhere on some Web server). This interface is sent as a message for the user.
3. The user selects the correct item from the message queue and the corresponding interface is shown in the display area. A set of buttons there defines the set of possible interactions with the application.

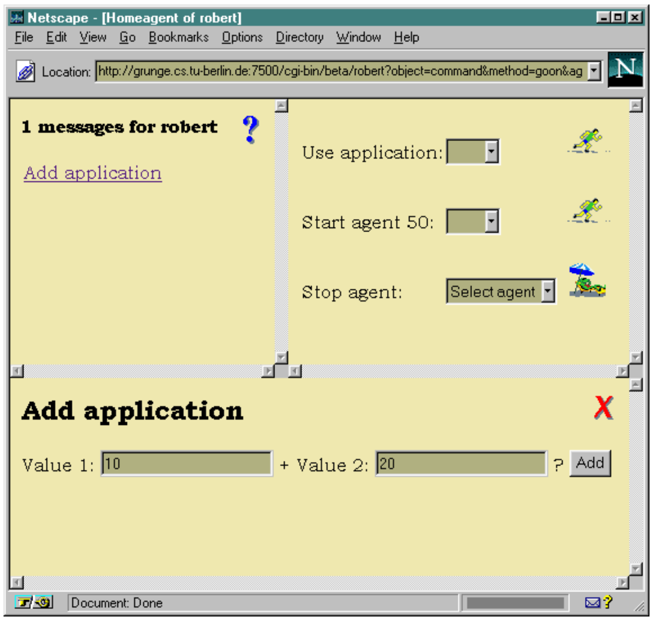

Fig. 2. A user interface agent

4. Selecting a button results in a request to the homeagent to use a specific method of the application agent. The homeagent issues a corresponding service request with the PageSpace coordination operations. The result of that interaction with an application agent is a string containing the new application interface generated by the agent, command, which is stored as a message for the user.
5. The user selects that message and is presented with the next interface, which includes the results of the previously-invoked operation.

Figure 3 shows the architecture of homeagents for the experimental prototype. It consists of three components that are implemented as different processes or threads: the interface for the interaction with the user which is visible as the generated user interface, the interface to the coordination kernel, and a repository where received messages are kept until they are deleted by the user

Applications in PageSpace are identified by a URL to a resource which contains a special application description. The use of a URL makes this approach wholly integrated with the Web. Paralelly to plain document-oriented Web, the agent-oriented PageSpace uses no central directory of applications.

Any application agent consists of three modules called Kernel, Service-Handler and Kernel-Connector. The Kernel is the “main-program” that loads and launches the module containing the provided services of the specific application, the Service-Handler. The Kernel-Connector provides the connection to

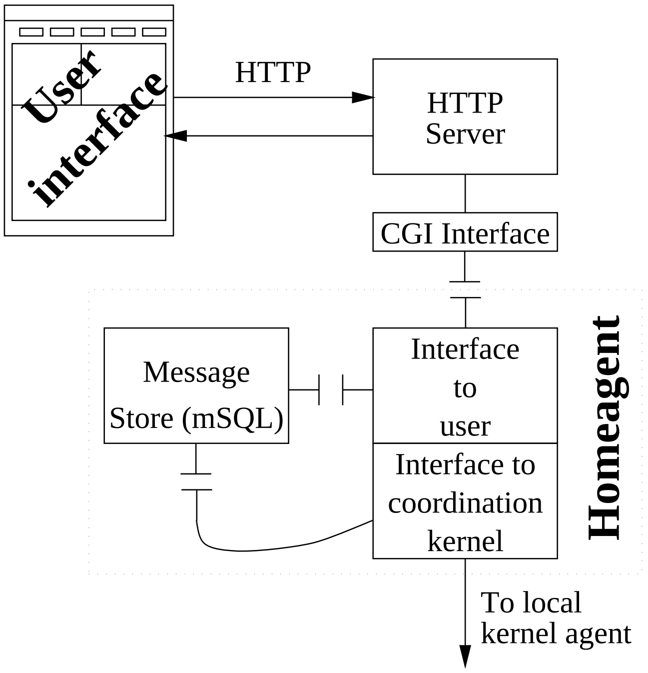

Fig. 3. The Architecture for the Homeagents

the Coordination Kernel of PageSpace by managing the connection to a Kernel agent and by propagating messages.

Kernel Agents are daemon processes that exist on specific machines and together spawn the Coordination Kernel, the virtual room for coordinating agents. Their functionality is central to the prototypical architecture of PageSpace. Kernel Agents contain the following modules:

- A *central control-unit* that invokes the kernel modules and executes them as Java threads. It constantly controls them by checking their states and taking appropriate action in the case of failures.
- A *server for homeagents* that is accessible via HTTP.
- The *Kernel-Connector* that establishes and manages the connections to other Kernel-Agents and implements the protocols for the distributed coordination kernel
- Some *Repositories* as the actual coordination technology used. They usually consist of a store of data items and methods to access them by coordination operations. In this prototypical PageSpace implementation, repositories for the coordination languages Laura (Tolksdorf, 1998) and Java/Linda are foreseen.
- The *Kernel-Connector* for the application and gateway Agents.

– The *Agent-Store* as a repository of application Agents that operate as threads within the Kernel-Agent. Application agents can be loaded by the Kernel and started as threads but do not necessarily have to. Gateway agents are always assumed to run external to the kernel.

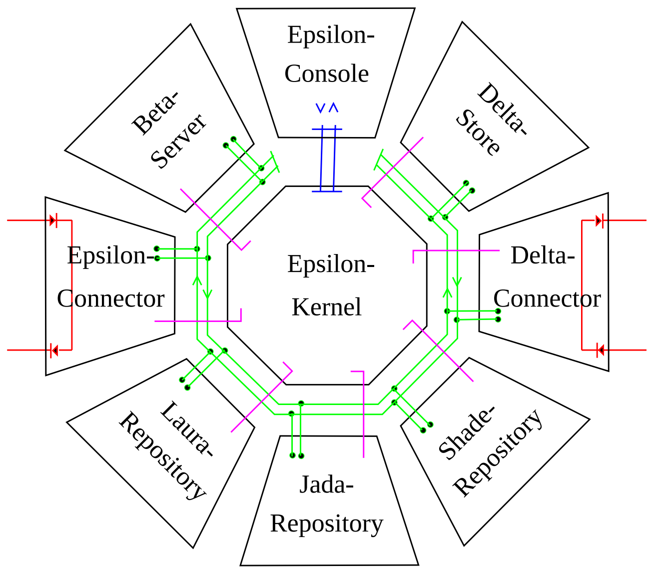

Fig. 4. The structure of the kernel agents

Figure 4 outlines the structure described. The initial prototype of PageSpace was implemented to evaluate our approach. Its implementation basis was Java 1.02 – with the RMI mechanism in Java 1.1 several technical details would change.

5 Building a PageSpace for cooperation: MUDWeb

In (Ciancarini und Rossi, 1997 we introduced Jada, a Java-Linda combination. Jada was a first experiment in combining Java with a coordination model. Here we describe JaCoPo, a combination of Java with a more expressive coordination model based on nested tuple spaces, as in Paradise.

JaCoPo is composed of a syntactic extension of the Java Language and a run-time environment. The syntactic extension enriches Java with a set of Linda-like coordination constructs. The run-time environment, implemented in Jada, supports multiple nested tuple spaces. The basic entity in a JaCoPo program is the *agent*. Each agent can access the available tuple spaces or create new ones.

Every agent has a *reference tuple space* that is used as the default shared space for the coordination operations that do not specify one.

A "move" operation is provided to allow an agent to change its reference tuple space at run-time, enabling what we call "migration by reference": although the agents do not move physically, their references for the outside world do, thus modifying the agent's environment. This is similar to what happens to Unix processes when they "move" across a file system changing their working directory.

In JaCoPo a *handle* identifies the root of a tuple space tree. Each tuple space tree is handled by a JaCoPoManager server named `root`. An agent can access a JaCoPoManager server using `Open`, then any *TupleSpace* managed by that server can be accessed using a path name as follows:

```
root # "ATupleSpace":"Subspace#1":"Subsubspace#2"
```

`"ATupleSpace"` is a subspace of `root`, `"Subspace#1"` is a subspace of `"ATupleSpace"`, `"Subsubspace#2"` is a subspace of `"Subspace#1"`.

All tuple space operations should specify a pathname. If no pathname is used, the operation refers to the default reference tuple space.

JaCoPo includes the basic *Linda-like* primitives: `Out`, `Read`, and `In`, either blocking or non-blocking. Moreover, it includes some set-oriented operations, like `rdall` and `inall`, whose result is a multiset of tuples stored in an array. For instance, the operation `rdall("message",?String:messages)` stores in array `messages` all strings found in the tuple space.

The JaCoPo runtime is based on two main classes, namely `JaCoPoManager`, which manages a tree of nested tuple spaces, and `JaCoPoObject`, which exploits the coordination services offered by a JaCoPoManager object.

5.1 MUDWeb

MUDWeb is a groupware platform we have designed and implemented using JaCoPo. We were inspired by MUD (Multi User Dungeon), a cooperative environment in which many people socialize and interact. MUDs can be used as enabling technologies for several groupware applications (Das et al., 1997, Rowley, 1997, Tarau et al., 1997, Doppke et al., 1998). A MUD platform usually offers the 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 an abstract platform that users can program to create shared virtual realities. Thus, a MUD is a very powerful abstraction to describe a general platform for cooperative work on the WWW, that provides a general framework for users to interact with each other and to manage items to be shared, such as documents and resources.

MUDWeb consists of a number of agents which coordinate themselves with respect to a set of services according to a number of protocols based on tuple exchanges. MUDWeb includes several "rooms" which correspond to tuple spaces including some default services.

A room is defined by a set of attributes: its name, some exits, some HTML code, some access rights, and some services. For instance, Fig. 5 shows the declaration of a room.

```
room JustARoom
access USER1, WIZARD
code "http://java/gaggi/nik/html/RoomCode.html"
service Counter "Services.CounterService"
    START\_VAL="1"
    HTML\_FILE="html/RoomCode.html"
exit OtherRoom1, OtherRoom2, Hall
data ("Tuple\_1", 1), ("Tuple\_2", 2, 2)
end
```

Fig. 5. A room in MudWeb

This room is named `JustARoom`. The clause `access` specifies that only users USER1 and WIZARD can enter the room. Clause `code` is mandatory, and describes the HTML code to be displayed when a user enters the room. Clause `service` specifies an agent named `Counter` being active in the room, offering service `Services.CounterService` (see below a description of the main available services). Clause `exit` specifies other rooms connected to this one, so that a user can "navigate" from room to room. Clause `data` specifies that this room contains two tuples.

The MudWeb architecture is implemented by a number of coordinated servers offering the basic abstraction of nested tuple spaces. Fig. 6 shows the software architecture of MUDWeb.

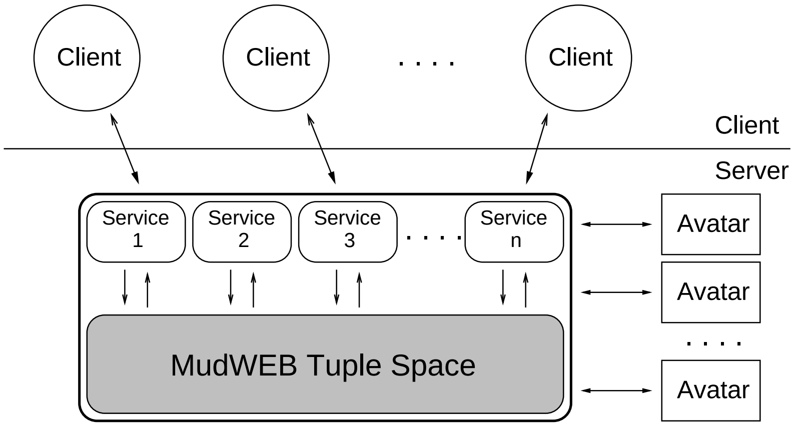

Fig. 6. A MUD-like active Web

Services are autonomous agents that wait for command tuples and perform services based on their content. Services are generally very small and specialized modules 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.

5.2 MUDWeb as a PageSpace

We have used JaCoPo as a coordination kernel, used by all agents to coordinate their activities. A MUDWeb server contains three kinds of agents: the *avatars*, the *services*, and the *mudshell*. The avatar is the persistent representation of a human user: it is a Homeagent controllable by the user through the Interface agent to be displayed within a browser. Actually an avatar can accept commands and return data in a variety of methods, including e-mail messages. Services are the Application agents in the shared space, that implement 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.

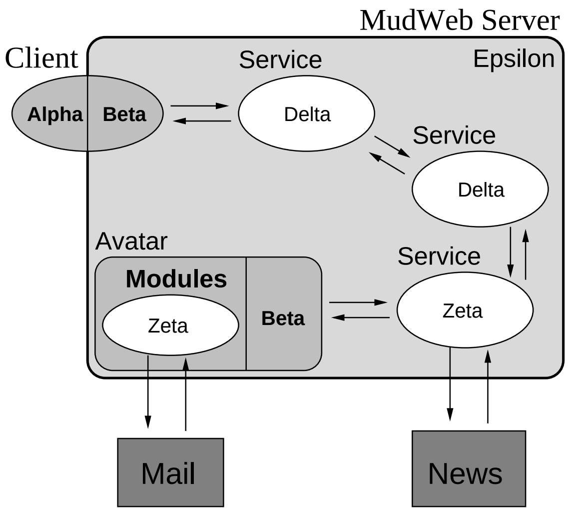

Fig. 7. MUDWeb as a PageSpace.

Comparing Fig. 6 with 7 we see how MUDWeb implements the PageSpace architecture.

The main innovation consists of how Homeagents are implemented. In MUDWeb a Homeagent is based on two different classes: the MudShell client and the Avatar. Both classes implement the MudPlayer interface, which define how people can interact with the architecture.

MUDWeb services are defined as agents operating in a room. Interestingly, some services are implemented as gateway agents. For instance, this is useful when a service has to input messages formed according to some special protocol like NNTP (news) and output their contents in some MUDWeb room. As another example, avatars exploit specific gateway agents to answer by e-mail to their users. An avatar can also emulate an HTTP server, because it has to be accessible by HTTP as well.

The design of a cooperative application based on MUDWeb consists of instantiating the classes of the framework showed in Fig. 8.

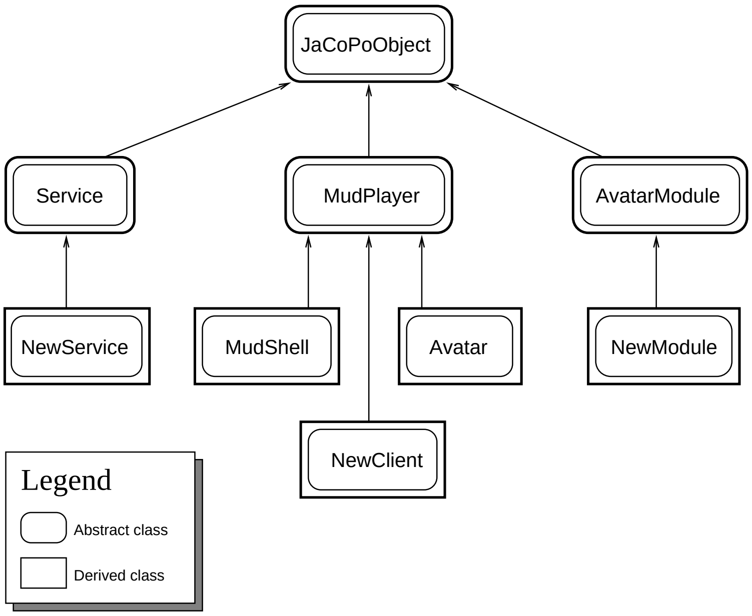

Fig. 8. The MUDWeb framework.

5.3 Conference management in MUDWeb

MUDWeb has been used to design and implement a workflow system supporting the management of a scientific conference. The system, named *ConfManager*, has the goal to simplify the management of the review of papers submitted to an international scientific conference.

The problem of managing the peer-review of papers submitted to a conference can be considered a case study in coordination. In fact, authors, reviewers, and program committee members can interact with the system using both synchronous (online WWW) and asynchronous (e-mail) communication interactions. Several activities can be automated, aiming at supporting the coordination protocols typical of a specific conference model.

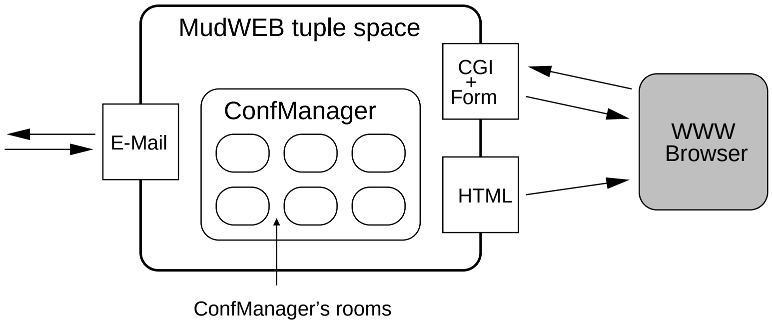

Fig. 9. Using MudWeb to design a Conference management System

ConfManager 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.

ConfManager offers in each room some specific services. All services come in two flavors, either synchronous or 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.

Services for authors of papers A new room is added when a new paper is submitted, and an avatar is created for the author(s). Submission is either online or by email: in both cases the command `submit` is used. Fig. 10 shows the format of an e-mail submission.

```
submit
Title: The World Wide Web as a Place to Work in
Authors: P. Ciancarini and R. Tolksdorf and F. Vitali
URL: http://hostname/click\_here
In this paper we describe our concept of an active Web,
namely how we design the software architecture of interactive
cooperative applications based on the WWW.  END
```

Fig. 10. Submitting a paper.

The method `processMail()` parses and interprets the message. The message has to specify the values for the title, authors, and URL fields on different lines. Other lines (eg. the abstract) are considered plain text to be simply translated in HTML and inserted in the room associated to the paper.

Another service that can be invoked by an author is `status`, in order to know some information on the review status (see below).

Services for reviewers Each *SubmittedPaper* room, created to store a submitted paper, will also store the reviews to it. 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 selection room.

A reviewer can work either on-line or by email. If the reviewer is working on-line he/she has to be in a paper's room. If the reviewer is using e-mail, he/she has to send a message `announce`, such as the one in Fig. 11.

```
announce
Paper: 1
Agent: reviewer1
```

Fig. 11. Announcing a review.

The message specifies an article id number and a username associated to the agent sending the message. A reviewer then fills in a form according to some evaluation grades. The filled form is sent to the reviewer avatar which transforms the message in a tuple to be stored in the paper room. Then the service `Services.Submitter` will be activated.

Services for PC members The service *Services.Selector* can be activated in the room *SelectRoom* and is implemented by the class `Services.Selector`.

This service is essentially a chat system to be used by PC members to talk and browse the stored dialogs.

6 Technology for Coordinating Web Applications

The PageSpace reference architecture is intended as a framework to help designers to deal with the coordination of distributed activities in Web based applications. It is interesting to compare our solution with alternative ones. One can distinguish different approaches studying where a Web application designer puts the main focus of activity in such applications, namely server-side, or client-side, or somewhere "in the middle", namely in the middleware.

In fact, the most conventional location of activity in Web based applications is at the servers side. This approach was taken very early in standard Web server technology by the introduction of the *Common Gateway Interface (CGI)*. According to CGI, some program can be executed upon the request of a URL, which dynamically generates an HTML page by performing some activity. The classical example for this approach is the generation of a report from some database contents, to be delivered to the requesting Web client in the form of an HTML page.

Mechanisms to preprocess additional tags within HTML pages appeared soon with early server software as "*server-side includes*". For higher expressiveness, complete languages were made available. The Web* system is a good example of such server sided activity is Web* (Almasi et al., 1995). Such a system interfaces a Web server with a Tcl/Tk interpreter, which in turn can access objects via CORBA. As a variant, the OreO approach implemented processing of information by using proxies (Brooks et al., 1995).

Meanwhile, commercial vendors provided complete server-based solutions for interactive and collaborative applications on the Web, such as Lotus Domino, or the systems based on Microsofts' active server pages.

In conclusion, today systems with server-side activity in applications are the dominant technology for Web applications. However, they inherit all problems of centralized systems, for example with respect to single point of failures, or performance bottlenecks.

A different option for designers of Web applications is to put the main focus of activity at the client side. With the first browser generations, only *helper* applications could be activated at the client side. A helper is an autonomous program callable from the browser but offering a different user interface, usually displayed in a different window: a typical example is the helper to display PostScript documents. Systems like SHAREDweb (Kumar et al., 1994) used helpers to provide application services, rather than displaying special media types. The coordination of multiple helpers became an issue; for example, the WWWinda system (Gutfreund et al., 1994) used Linda-like coordination technology to coordinate a browser orchestrating multiple helpers to be synchronized for multimedia documents.

It became rapidly clear that it was desirable to integrate an execution environment *within* the browser. A variety of prototypical browsers was developed for various languages, such as Objective Caml (Rouaix, 1996), or Python, (see the Grail browser at `http://grail.cnri.reston.va.us/grail/tcl`), etc.

The dominance of Java and JavaScript, incorporated in the two major Internet browsers, has pretty much satisfied the "market" for client-side activity. However, despite the initial hopes, both are currently not widely used for distributed applications. Java can be used to design highly "intelligent terminals" to access special applications – an example is Internet banking software, such as Twister from Brokat (see `www.brokat.de`). Instead, JavaScript seems to be used only to enhance the user interface of Web pages, hardly ever more than "highlighting" parts of pages by exchanging displayed images when the mouse pointer moves across them.

HTTP as the communication protocol in the Web makes interactive applications very complicated to implement. This is mainly due to the connectionless, unidirectional paradigm embedded in HTTP. Activity could be coordinated by extensions to the communication protocol.

With the advent of plugins, this option has been used to establish communication channels between plugins and proprietary servers, that go beyond the limitations of HTTP. The HTTP-NG protocol currently developed by the World Wide Web Consortium will also contain extensions to allow coordination of activity at the server and the client side. Currently, however, HTTP provides the communication platform for the Web and it is unclear which browser generations will support more sophisticated protocols.

One could also understand HTTP as a pure transport protocol and stick to existing middleware environments to coordinate open systems for applications. First prototypes used CORBA (Rees et al., 1995) and DCE (Beitz et al., 1995) as the actual implementation basis for open applications.

Soon, commercial vendors have developed systems and specifications that allow for interaction of distributed components and objects via the Web. Examples are RMI from Sun within the Java standard, DCOM from Microsoft, and IIOP, an Internet integration of CORBA that can be found, for example in Netscapes Java libraries giving access to OMG compliant object brokers.

Today, there seems enough of such enabling technology implemented and available within Web technology. However, there is still no "killer application" in widespread use that turns Internet users into participants in a distributed world of active objects or agents. Our proposal, namely PageSpace, defines a design framework and offers some enabling tools to help designers of coordinative, agent-based Web applications.

7 Conclusions and Future Work

Our initial vision of PageSpace had to be refined in the course of designing and implementing the platform. During the project we made several observations on applying coordination technology to the Web. In fact, the issue of coordination in Web applications is still an open question which is not settled in a satisfactory manner. The Web can be the environment in which coordination technology could prove its strength and provide a convincing example of its usefulness of because it has the potential for the integration of an active, agent-based Web by a uniform notion of activity and coordination. Interestingly, the family of Linda-like coordination languages we have developed is only one possible flavor of coordination technology, and it stands in direct competition with other, more industrial technologies, like distributed, heterogeneous object-oriented platforms à la CORBA. In the field of Web applications, coordination technology has to compare with approaches that at moment have a much stronger industrial support and service resources. The implementation of coordination technology in the form of a programming language to control agents over the World Wide Web faces severe engineering problems with respect to the organization of shared coordination media to be used, their replication, the addressing of elements therein, and handling potentially high load demands.

At the moment we are exploring the interaction capabilities offered by Linda-like coordination models to mobile Java agents. Another issue we are exploring is security, that is especially important for e-commerce applications based on PageSpace (Ciancarini et al., 1997, Rodriguez et al., 1997). We also are studying how the advent of XML technology can be exploited for agent-based programming. More information on current activities in PageSpace can be found on the Web at `http://www.cs.tu-berlin.de/~pagespc`.

Acknowledgments. PageSpace has been supported by the EU as ESPRIT Open LTR project #20179. It is currently supported by the German DAAD and the Italian CNR within the VIGONI program. Several people contributed to the development of the ideas described in this paper; we thank here especially the students G. Busi and N. Gaggi of UniBologna, and A. Knoche of TU Berlin.

References

Adler, R. (1995). Distributed Coordination Models for Client/Server Computing. *IEEE Computer*, 28(4):14–22.

Almasi, G. et al. (1995). Web*: A Technology to Make Information Available on the Web. In *Proc. 4th IEEE Workshop on Enabling Technology: Infrastructure for Collaborative Enterprises*, pages 147–153, Berkley Springs, WV. IEEE Computer Society Press.

Bannan, J. (1997). *Intranet Document Management*. Addison-Wesley.

Beitz, A. et al. (1995). Integrating WWW and Middleware. In Debreceny, R. und Ellis, A., editors, *Proc. 1st Australian World Wide Web Conference*, Lismore, NSW. Norsearch Publishing.

Bjornson, R., Carriero, N., und Gelernter, D. (1997). From weaving threads to untangling the web: a view of coordination from Linda's perspective. In Garlan, D. und LeMetayer, D., editors, *Proc. 2nd Int. Conf. on Coordination Models and Languages*, volume 1282 of *Lecture Notes in Computer Science*, pages 1–17, Berlin, Germany. Springer-Verlag, Berlin.

Brooks, C., Mazer, M., Meeks, S., und Miller, J. (1995). Application-Specific Proxy Servers as HTTP Stream Transducers. In *Electronic Proc. 4th Int. World Wide Web Conference "The Web Revolution"*, Boston, MA.

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

Ciancarini, P. (1996). Coordination Models and Languages as Software Integrators. *ACM Computing Surveys*, 28(2):300–302.

Ciancarini, P., Knoche, A., Rossi, D., Tolksdorf, R., und Vitali, F. (1997). Coordinating Java Agents for Financial Applications on the WWW. In *Proc. 2nd Int. Conf. on Practical Applications of Intelligent Agents and MultiAgent Technology (PAAM)*, pages 179–193, London, UK.

Ciancarini, P. und Rossi, D. (1997). Jada: Coordination and Communication for Java agents. In Vitek, J. und Tschudin, C., editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 213–228. Springer-Verlag, Berlin.

Ciancarini, P. und Rossi, D. (1998). Coordinating Java Agents Over the WWW. *World Wide Web*, 1(2):87–99.

Ciancarini, P., Tolksdorf, R., Vitali, F., Rossi, D., und Knoche, A. (1998). Coordinating Multiagent Applications on the WWW: a Reference Architecture. *IEEE Transactions on Software Engineering*, 24(5):362–375.

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

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

Gutfreund, Y., Nicol, J., Sasnett, R., und Phuah, V. (1994). WWWinda: An Orchestration Service for WWW Browsers and Accessories. In *Proc. 2nd Int. World Wide Web Conference*, Chicago, IL.

Kumar, V., Glicksman, J., und Kramer, G. (1994). A SHAREd Web To Support Design Teams. In *Proc. 3rd IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Morgantown, WV.

Lewis, T. (1995). Where is Client/Server Software Headed? *IEEE Computer*, 28(4):49–55.

Munson, J. und Dewan, P. (1997). Sync: a Java Framework for Mobile Collaborative Applications. *IEEE Computer*, 30(6):59–66.

Orfali, R., Harkey, D., und Edwards, J. (1996). *The Essential Distributed Objects Survival Guide*. Wiley.

Rees, O., Edwards, N., Madsen, M., Beasley, M., und McClenaghan, A. (1995). A Web of Distributed Objects. *World Wide Web Journal*, 1(1):75–88.

Rodriguez, J., Noriega, P., Sierra, C., und Padget, J. (1997). FM96.5 A Java-based Electronic Auction House. In *Proc. 2nd Int. Conf. on Practical Applications of Intelligent Agents and MultiAgent Technology (PAAM)*, pages 207–224, London, UK.

Rouaix, F. (1996). A Web navigator with applets in Caml. *Computer Networks and ISDN Systems*, 28(7-11):1365–1371.

Rowley, M. (1997). Distributing MOO-Based Shared Worlds. In *Proc. 6th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 155–160, Boston. IEEE Computer Society Press.

Tarau, P., Dahl, V., und DeBosschere, K. (1997). A Logic Programming Infrastructure for remote Execution, Mobile Code and Agents. In *Proc. 6th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 106–112, Boston. IEEE Computer Society Press.

Tolksdorf, R. (1998). Laura – A Service-Based Coordination Language. *Science of Computer Programming*, 31(2-3):359–382.