The World Wide Web as a Place to Work in

P. Ciancarini and D. Rossi and F. Vitali
Dipartimento di Scienze dell'Informazione
University of Bologna - Italy
E-mail: {ciancarini,rossi,vitali}@cs.unibo.it

Abstract

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. An active Web includes agents able to use the services offered by WWW clients and servers. In an active Web both users and agents can interoperate using a set of basic services for communication and synchronization. The active Web we describe here is based on coordination technology: we explore two alternative implementations, one based on a platform called Paradise and another fully based on Java.

1. Introduction

The World Wide Web has the potential to become the standard platform to access Internet services, being the most widespread architecture to build integrated applications. In fact, most application domains are turning to the World Wide Web as the environment of choice for building innovative applications leveraging on the open standards, their diffusion, and the programmable nature of the environment. Thus, 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 [**?**].

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 sophisticated agent coordination.

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). 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 [**?**] 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 positions. The PageSpace architecture is based on the Linda coordination model [**?**] 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 Stock Exchange is an application for performing multiple distributed auctions of goods, using the WWW as the underlying infrastructure and Paradise as its coordination language. Instead, MUDWeb is a MUD-like environment for creating collaborative applications on the Web. It uses JaCoPo, an extension of Java, as its coordination language. Both application use the PageSpace architecture as the underlying design framework for their implementation.

The paper is structured as follows: in section **??** we discuss the issues arising when creating distributed, interactive applications on the WWW. In section **??** we illustrate the overall PageSpace architecture. In section **??** we discuss implementing a PageSpace application using Paradise as the coordination language, and illustrate the Stock Exchange application. In section **??** we discuss the JaCoPo coordination language, and illustrate the MUDWeb application. Section **??** concludes the paper.

2. Designing active Web applications

The Word Wide Web 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 Word Wide Web 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” the 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.

2.1. The WWW as an agent world

An *active Web* is a system including some notion of agent performing some activity. 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, decision support system or expert system) or even at the user level. An active Web includes several *agents*, with a well-defined behavior. Each component of an active Web is as an autonomous agent in a world offering coordination 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 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 behavior) 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: this schema fits quite well into the distributed objects model [**?**]. In our approach agents perform sequences of 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.

We distinguish between agent computation and agent coordination.

Agent computation is what concerns the internal behavior of an agent.

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 [**?**].

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 senses. First and foremost, it offers very simple coordination entities, 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, we say that 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 a Jada, an extension of Java [**?**].

3. The PageSpace

PageSpace is a reference architecture for multiagent applications built on top of the WWW [**?**]. PageSpace applications consist of a number of distributed agents to be coordinated to accomplish some cooperative task.

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

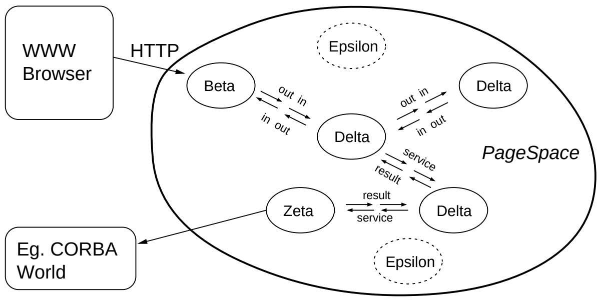

Figure 1. The PageSpace reference architecture

- *Alpha* or *user-agents*, are user interface agents. Alpha agents consists of applets, scripts, and HTML pages.
- *Beta* or *home-agents*, are a persistent representation of users on the PageSpace. They allow the users to access the application agents on the PageSpace, provide the user with the required Alpha, and collect the messages addressed to the alpha agent or the user in their absence.
- *Gamma* is the coordination architecture used for the specific implementation of PageSpace. Examples of Gamma can be coordination languages such as Paradise [**?**] or Jada [**?**].
- *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.
- *Epsilon*, or *core agents* are the functional core of the PageSpace architecture for every computer it is running on. There is exactly one Epsilon agent on each machine taking part to the PageSpace. Epsilon agents take care of activating and destroying agents, provide access to the PageSpace by external agents via HTTP servers, and provide coordination mechanisms among 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 shows al least one Alpha agent, which is connected and interacts directly with a Beta, running on the server and directly connected to the PageSpace. A set of Delta agents implement the functionalities of the distributed application. Zeta agents provide access to external environments such as Corba-based applications. The PageSPace is implemented by a set of Epsilon agents that give body to the Gamma environment where the Delta agents reside and interact. Delta and Zeta agents can reside on any machine taking part to the PageSPace, while Alpha reside on the client, Beta on a specific machine taking part t the PageSpace, and Epsilon agents are present one every machine taking part to the PageSpace.

4. Building a PageSpace using Paradise

We present here an implementation of the PageSpace based on Paradise, a programming environment inspired by Linda. We used this implementation to test a WWW- based electronic-commerce application (a stock exchange application).

4.1. Paradise

Paradise is the version of Linda for coordinating open distributed systems [**?**]. We recall that in Linda coordination is achieved using a medium called *tuple space*. Agents are usually sequential processes (eg. they are written in C or FORTRAN) that use special operations to access the tuple space. These operations allow an agent to add an active/passive tuple or to read/remove a tuple using an associative addressing scheme.

Paradise adds support for possible persistent multiple tuple spaces. Agents can operate transactions across tuple spaces. The tuple spaces (that can be dynamically created) implement an access restriction policy (so that, for example, only authorized agents can remove tuples from a tuple space) that is similar to the one used in Unix filesystems.

We present an example to show how a simple groupware system, namely an e-mail distribution system, could be designed using Paradise. Suppose that a little company owns only one mailbox visible from outside, whereas each company employee has a personal internal mailbox.

We intend to design a system able to distribute mail from the global mailbox to the private ones. We use a tuple space for the global mailbox, and one tuple space for each private mailbox (fig. **??**).

The Paradise program shown in **??** implements the software architecture for PostOffice.

This program first opens a persistent tuple space named *rootts*; it is a service space which represents the basic Paradise abstraction (the Paradise server) used to create new tuple spaces.

Then, operation `create` creates a persistent tuple space (a persistent tuple space survives even after the Paradise

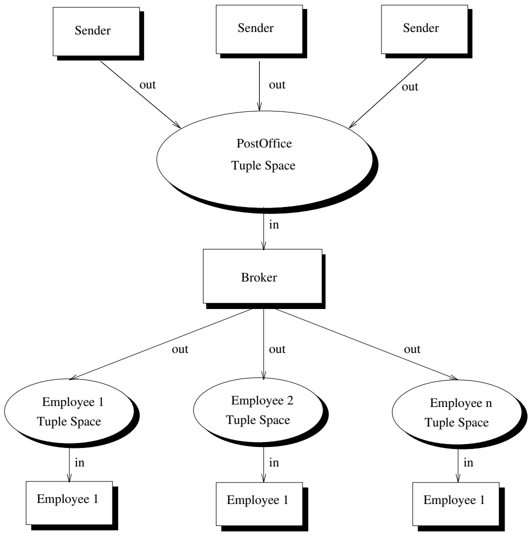

Figure 2. Post Office example

```
/*************************************
PROGRAM 1
Create a persistent PostOffice
and several LetterBox tuple spaces
**************************************/
#include <paradise.h>

main() {
  TSHANDLE myts, rmyts;

  open @ rootts();

  /* Create Post Office tuple Space */
  myts = create @ rootts(PARADISE_PERSISTENT,
         PARADISE_LABEL("PostOffice"));
  rmyts = restrict @ myts(PERM_OUT);
  register_handle("PostOffice","Sender","xxxx", &rmyts);
  rmyts = restrict @ myts(PERM_IN);
  register_handle("PostOffice","Broker","xxxx", &rmyts);
  close @ myts();

  /* Create a LetterBox for each employee */
  myts = create @ rootts(PARADISE_PERSISTENT,
         PARADISE_LABEL("LetterBox Employee 1"));
  rmyts = restrict @ myts(PERM_OUT);
  register_handle("Employee 1","Broker","xxxx", &rmyts);
  rmyts = restrict @ myts(PERM_IN);
  register_handle("Employee 1","Employee","xxxx", &rmyts);
  close @ myts();

  /* Create a LetterBox for unknow employee */
  myts = create @ rootts(PARADISE_PERSISTENT,
         PARADISE_LABEL("LetterBox Unknown"));
  rmyts = restrict @ myts(PERM_OUT);
  register_handle("Unknown","Broker","xxxx", &rmyts);
  close @ myts();

  close @ rootts();
}
```

Table 1. Creating the PostOffice software architecture

```
/************************************
PROGRAM 2
Insert a new message into PostOffice
*************************************/
#include <paradise.h>
main(){
  TSHANDLE myts;
  lookup_handle("PostOffice", "Sender", &myts);
  open @ myts();
  out @ myts("Employee 1", "Hello!");
  close @ myts();
}
```

Table 2. Sending a message

program terminates) labelled PostOffice. Such an operation requires a *handle*, namely a reference to an open tuple space. It also returns a handle (`myts`) that can be used to access the new tuple space.

Paradise agents use handles to access tuple spaces; names are reserved to the Paradise admin, who uses them to access the Paradise server in case of problems.

A handle can be redefined as a *restricted tuple space handle* using `restrict`. The agent creating a tuple space has full access, namely it can write, read, and delete tuples. The `restrict` operation creates a new handle to the same tuple space using a different set of access rights. In our example, the handle Sender is registered only for `out`, while the handle Broker is registered only for `in`.

The registration of handles is managed by the Paradise server, which exports the available handles to all agents. Registration happens as follows: `register_handle(name1, name2, password, ts_handle)` where `name1`, `name2`, and `password` are strings whereas `ts_handle` refers to a tuple space. Strings name1 and name2 implement a two-layers name space; the proposed password is only needed when the handle must be deleted.

The program in Tab. **??**) shows how a sender works. Function `lookup_handle` requires the names to identify a handle and returns the handle as `&myts`, if found.

A broker is then necessary to read messages from the PostOffice and to deliver them to the private mailboxes. Our implementation of the broker uses transactions. Transactions in Paradise are used to group several operations as a unique atomic operation in which either all or none succeed.

The operation `xaction` starts a transaction, while the operation `commit` terminates it. Once a transaction is started there are two possible results: either the transaction is permanently recorded or it is aborted. An agent can explicitly abort a transaction using the `cancel` operation, but only if the transaction has not been concluded yet with a commit (*pending transaction*).

The broker works as follows: after having obtained a handle, the broker agent waits for a message for any re-

```
/****************************************
PROGRAM 3
Broker using transactions
****************************************
#include <paradise.h>

main() {
  TSHANDLE myts, ts;
  char addressee[100];
  char message[256];
  int stat, num;

  lookup_handle("PostOffice", "Broker", &ts);
  open @ ts();
  while (TRUE){
    /* Read message from "PostOffice" tuple space */
    in @ ts(?addressee, ?message);

    if (lookup_handle(addressee, "Broker", &myts) == 1) {
      open @ myts();
      /* Put the message in the "Employee" tuple space */
      xaction @ myts();
      in @ myts ("Counter", ?num);
      out @ myts(num, addressee, message);
      out @ myts ("Counter", num+1);
      commit @ myts();
      close @ myts();
    } else {
      lookup_handle("Unknown", "Broker", &myts)
      /* Put message in the "Unknown" tuple space */
      open @ myts();
      out @ myts(addressee, message);
      close @ myts();
    }
  }
}
```

Table 3. A broker agent using transactions.

ceiver with the operation `in @ myts(?addressee, ?message)`. When a message is put in the tuple space, the operation succeeds and the message is delivered to the addressee's private mailbox; if the address is unknown then a default tuple space is used for delivery.

We also count and enumerate the messages for each employee. A possible solution consists of inserting a counter in each private mailbox. The counter tuple is the tuple `("Counter", num)`, where `"Counter"` is a string, while `num` is an integer.

Table **??** represents the broker agent using transactions to enumerate messages.

The broker agent reads the counter tuple, stores the message giving its number, then stores a new incremented counter tuple.

4.2. The Stock Exchange

The Stock Exchange is a test application we built with our Paradise-based PageSpace. The aim of this application is to test PageSpace in the Electronic Commerce field.

Currently, most e-commerce applications on the WWW can be classified as virtual malls. A *virtual mall* is like a shop in which a user can enter, examine the items on sale, and buy some of them. The actual delivery of the items is performed by a shipping service in the real world.

From a technical point of view virtual malls differ in the proprietary interaction schemata they use. Thus, each virtual mall is a component of its own on the WWW. This mirrors what happens in the real world: different shops in different places need different people to manage each shop. However, such heterogeneity is a drawback on the WWW: on each site buyers have to check what kinds of payment forms are accepted, which security measures are guaranteed, which privacy level is assured, and so on.

We developed a Stock Exchange application as a first step towards an integrate infrastructure to support Electronic Commerce applications based on the WWW.

In this application our items on sale are those vegetables for which an EC quality recommendation exists. In fact the application has been developed according to the specifications given by the managers of Bologna's Food Gross Market.

The basic requirements for the application were:

- the interface must be able to show one or more pictures of the product;
- the interface must be able to show a quality label for each product and must provide access to the complete description of the quality recommendations.

The application must support two major interaction paths:

- direct: the user directly manages exchanges;
- via software agents: the user delegates software agents to manage exchanges in an autonomous way but under the control of general behavior rules.

This application gives its users the ability to perform some buy/sell operations; here is a rough list of the services offered:

- give the list of pending ”buy” offers for a product;
- give the list of available ”sell” offers for a product;
- submit a “buy” or “sell” offer;
- “buy” or “sell” query;
- undo “buy” or “sell” query.

The design of the application architecture is based on Pagespace (some ad-hoc changes were necessary in order to adapt to the application requirements). The first step is to indentify alpha, beta, gamma, and delta. Alpha agents are, of course, the user interfaces in WWW browsers; beta agents are the browser companion processes that act like bridges between the user interface and the coordination layer; delta

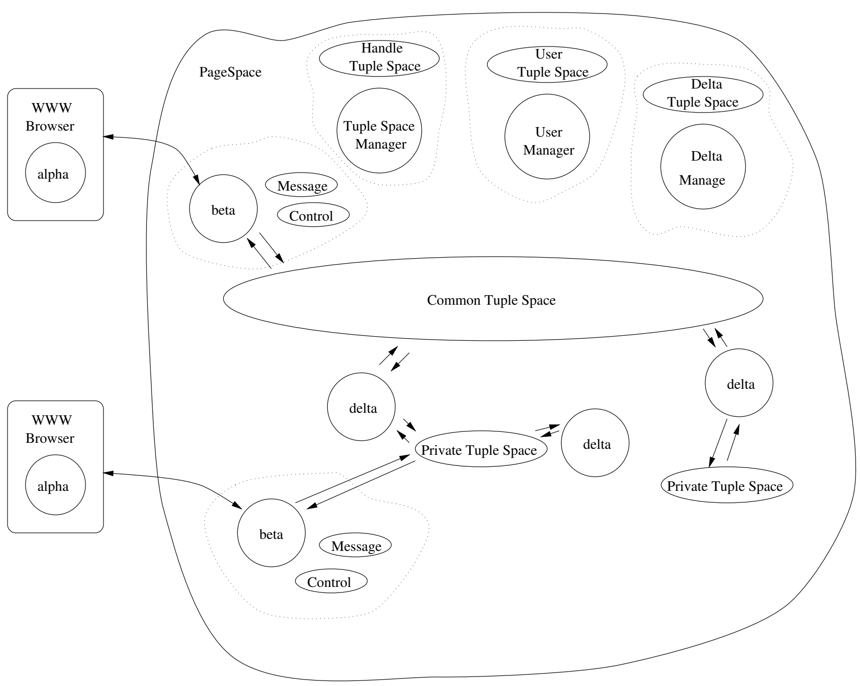

Figure 3. PageSpace with Paradise

agents are the service providing components used by the Stock Exchange manager in order to create a virtual stock market environment; the gamma environment is, of course, Paradise.

All agent interactions are based on tuple exchanges using private tuple spaces as channels. Most interactions follow the following protocol:

- the first agent puts a tuple in second agent's tuple space asking for an operation;
- the second agent performs the requested operation and puts the resulting data, in form of tuples, into the first agent's tuple space.

All these operations are performed using distributed transactions in order to avoid inconsistencies due to network lag or host failures.

4.3. The Stock Exchange application as a PageSpace

We have used Paradise as a Gamma coordination environment, used by all agents to coordinate their activities. Using multiple tuple spaces we built a modular system in which tuple spaces are used as both the communication channels and the coordination media.

In fig. **??** we depict the architecture of the PageSpace as written in Paradise. When the system boots up the Tuple Space Manager (an Epsilon agent) is executed; it creates the Handles' Tuple Space, the Common Tuple Space and it activates the remaining Epsilon agents.

All the other tuple spaces are created (and possibly deleted) on the fly by agents. The Tuple Space Manager is used to register each new tuple space in order to keep track of what is going on in the system.

The Common Tuple Space is used by agents in order to find each other; once they join in the Common Tuple Space they can dynamically create a new private tuple space.

The User Manager takes care of managing PageSpace's users. The first time a user logs into the PageSpace she accesses an HTML page where, using forms, she writes username, password, and other information. The CGI-BIN script activated by posting the form sends the relevant information to the User Manager. This returns a pointer to the user's Beta agent, creating it if necessary.

5. Building a PageSpace using Java+Linda

In [**?**] we introduced Jada, a Java-Linda combination. Jada was a first experiment in combining Java with a coordination model. Here we describe another combination of Java with a coordination language called JaCoPo that implements 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 has been implemented in order to provide Java a set of Linda-like coordination constructs that can act on multiple tuple spaces (similarly to Jada, which JaCoPo uses as its coordination kernel).

The basic entity in a JaCoPo program is the *agent*. Each agent can access the tuple spaces available on the system 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 the agent to change its reference tuple space at run-time, enabling what we call a “migration by reference” of the agents: 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 which “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`. Agents can then access a *TupleSpace* using a path name:

```
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"`.

Pathnames are used in the code of agents when invoking the primitives to access a tuple space. 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`, both blocking and non-blocking. Moreover, it includes some set- oriented operations, like `rdall` and `inall`, whose result is usually a multiset of tuples which is stored in an array. For instance, the operation `rdall("message", ?String:messages)` will store in array `messages` all strings found in the tuple space.

The JaCoPo runtime is based on two main classes, namely `JaCoPoManager`, which manages a nested multispace extending the `ObjectServer` class in Jada and offering a coordination service to the `JaCoPoObject` class, which extends `ObjectClient` in Jada, which in turn extends Java’s Applet class.

5.1. MUDWeb

MUDWeb is a test application we built with a JaCoPo-based PageSpace. The aim of this application is to test PageSpace in the field of cooperative work.

A MUD (Multi User Dungeon) is a cooperative environment in which many people socialize and interact; MUDs can be used as enabling technologies for some kinds of groupware applications [**?**, **?**, **?**]. A MUD usually represents the infrastructure of a role-playing game (hence the

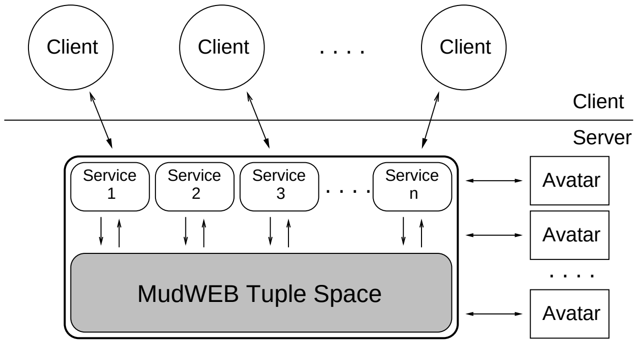

Figure 4. A MUD-like active Web

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 creates 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, for users to interact with virtual items (such as documents).

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. The whole architecture is implemented by a server organized as nested tuple spaces **??**.

Fig.**??** shows the software architecture of MUDWeb.

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

For instance, an application we have implemented is a workflow system supporting the management of 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.

5.2. MUDWeb as a PageSpace

We have used JaCoPo as a Gamma coordination environment, used by all agents to coordinate their activities.

A server provides the functionalities of an Espilon agent. Within such a server three kinds of agent are active: the *avatars*, the *services*, and the *mudshell*. The avatar is the persistent representation of a human user. Thus it is a Beta

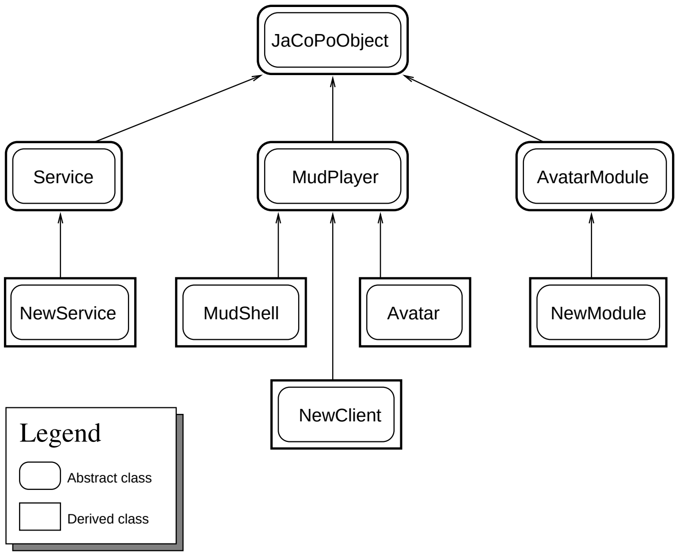

Figure 5. The MUDWeb framework.

agent performing for the user. The avatar provides the Alpha 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.

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

6. Conclusions and Future Work

We have described two implementations of PageSpace, based on two coordination languages. The first implementation is based on Paradise, whereas the second is all pure Java.

The two implementations are similar insofar as they exploit the same coordination model, namely multiple tuple spaces plus Linda-like operations. They are also different insofar as the second one is completely pure Java.

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 [**?**, **?**].

More information on current activities in PageSpace can be found on the Web at `http://www.cs.unibo.it/˜cianca`.

Acknowledgments. PageSpace has been supported by the EU as ESPRIT Open LTR project #20179 as a joint work with Robert Tolksdorf of TU Berlin. Several people in our department contributed to the development of the ideas described in this paper; we thank here especially our students Gianluca Busi and Nicola Gaggi.