Coordinating Multiagent Applications on the WWW: A Reference Architecture

Paolo Ciancarini, *Member*, *IEEE Computer Society*, Robert Tolksdorf, *Member*, *IEEE Computer Society*, Fabio Vitali, Davide Rossi, and Andreas Knoche

Abstract—The original Web did not support multiuser, interactive applications. This shortcoming is being studied, and several approaches have been proposed to use the Web as a platform for programming Internet applications. However, most existing approaches are oriented to centralized applications at servers, or local programs within clients. To overcome this deficit, we introduce PageSpace, that is a reference architecture for designing interactive multiagent applications. In this paper we describe how we control agents in PageSpace, using variants of the coordination language Linda to guide their interactions. Coordination technology is integrated with the standard Web technology and the programming language Java. Several kinds of agents live in the PageSpace: user interface agents, personal homeagents, agents that implement applications, and agents which interoperate with legacy systems. Within our architecture, it is possible to support fault-tolerance and mobile agents as well.

Index Terms—Distributed programming systems, Java, Linda, coordination, Internet, Web applications, open distributed systems.

---

1 Introduction

The Web has evolved into the dominating software architecture for information systems on the Internet. There is increasing demand to use it as a platform for programming distributed applications in which processing of information occurs. For instance, the application domains of project and workflow management [1] and electronic commerce [2] include classes of applications that require distributed access and processing due to the distributed nature of the work these applications support.

Currently there is no widely accepted reference architecture for implementing interactive distributed applications on top of the Web. Web browsers supporting Internet programming languages such as Java allow activity at user interface level in the form of *applets*. However, languages like Java need integrated middleware (e.g., CORBA) to coordinate activities tied to multiple, distributed clients [3]. Coordination has to be centralized at some server to which all users participating in an application have to connect to. Thereby, the activity located at the browser does not really make the application distributed, as applets at the browser cannot connect to other applets providing services to them directly. In fact, providers of Java technology are developing middleware in the form of Java libraries called Java RMI (Remote Methods Invocation), to interface (new) remote applications written in Java as well, and JavaIDL (Interface Description Language), to interface via CORBA legacy applications written with other languages.

We have developed an original solution to this problem. In fact, the PageSpace [4] is a reference architeture to support distributed applications on the WWW. It is based on the core Web technology for access and presentation, on Java as the execution mechanism, and on *coordination technology* [5] to manage the interaction of agents in a distributed application. This paper describes the rationale of PageSpace, its design, and the implementation strategies currently applied.

Currently, the field of electronic commerce is of particular interest for the application of platforms like PageSpace. In fact, electronic commerce should serve as a benchmark for the validation of our approach wrt. applications requirements. Section 4.6 describes a case study.

This paper is organized as follows. In Section 2.1 we review the main approaches to implement applications that require active processing on the Web. In Section 2.2 we then describe our specific approach to coordination of distributed applications. Section 3 describes the PageSpace and the various kinds of agents it includes. Then, in Section 4 we outline our approach in engineering and implementing PageSpace and describe a case study. Finally in Section 5 we compare our approach to a number of alternative solutions introduced to design interactive multiagent WWW applications.

2 Programming the Web

Since 1993 the Web as the dominating Internet service has evolved into the most popular and widespread platform for world wide accessible information systems. The software for accessing and offering information on the Web is available in the public domain for all hardware and operating system platforms in use.

2.1 Existing Approaches for Programming the Web

At its core, the Web is a static hypertext graph in which documents marked up in HTML are offered by servers,

---

- *P. Ciancarini, F. Vitali, and D. Rossi are with the Department of Computer Science, University of Bologna, Via Mura A. Zamboni, 7, I-40127 Bologna, Italy. E-mail: {cianca, vitali, rossi}@cs.unibo.it.*
- *R. Tolksdorf and A. Knoche are with the Technische Universität Berlin, Fachbereich 13, Informatik, FLP/KIT, FR 6–10, Franklinstr. 28/29, D-10587 Berlin, Germany. E-mail: {tolk, knoche}@cs.tu-berlin.de.*

Manuscript received 1 July 1997; revised 17 Dec. 1997.
Recommended for acceptance by G.-C. Roman and C. Ghezzi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 106412.

0098-5589/98/$10.00 © 1998 IEEE

retrieved by clients with the HTTP protocol, and displayed by graphical interface that is very easy to use. Because of its diffusion, it is desirable to use the Web as a platform for dynamic, distributed applications. The support offfered by the core Web platform for applications is very rudimentary —only the CGI mechanism allows for processing of information that is entered by the user in forms, or retrieved from auxiliary systems, such as database servers.

Several mechanisms have been proposed in order to make the Web a more effective platform for multiuser distributed applications. The following classification is structured according to the loci of activity where these mechanisms act.

2.1.1 Activity Located at Web Servers

The CGI mechanism can be used to access other application servers from the Web. A typical example is database access, where some form allows the formulation of a query in a browser and a CGI script at the server passes that query—probably in a translated form—to some database server. The results of the query then are converted to HTML and sent back to the users browser. Fig. 1a shows the structure of such a distribution of activity.

This approach turns out to have nothing in common with distributed paradigms like client-server interaction. In fact, interfacing an application via CGI to the Web does not mean to offer a distributed application. There is no processing at the client besides displaying results. Moreover, there is only one central location of activity—the server. Thus, such an application is basically a mainframe/ terminal system on the Internet. The Web server is comparable to a mainframe—the only location of processing. The Web browsers are nothing but graphical, easy-to-use terminals, interpreting HTML as the display language.

2.1.2 Activity Located at Web Clients

When a Java applet is executed within the browser, again it usually performs no distributed application. The applet is just a program that is run locally on the user's machine. There is no generally accepted way to connect applets running on different machines; the remote method invocation (RMI) mechanisms lead to security problems that are not solved yet. Some applets and plug-ins—such as RealAudio players—connect to other proprietary servers and thereby abandon core Web technology.

Fig. 1b shows this structure of activity focused on clients, which contains no generally accepted framework for distributed applications on the Web.

2.1.3 Activity Located in Middleware

A third approach to distributed applications on the Web is to use middleware to connect the active parts in an application, which can be located in clients and/or servers. Here, the Web technology takes the role of providing a uniform access and presentation mechanisms. Fig. 1c depicts that structure. The paper [6] discusses which kind of coordination interactions need modern distributed applications. These include direct/indirect service requests, explicit/implicit invocations of multiple servers, blocking/nonblocking primitives for receiving responses.

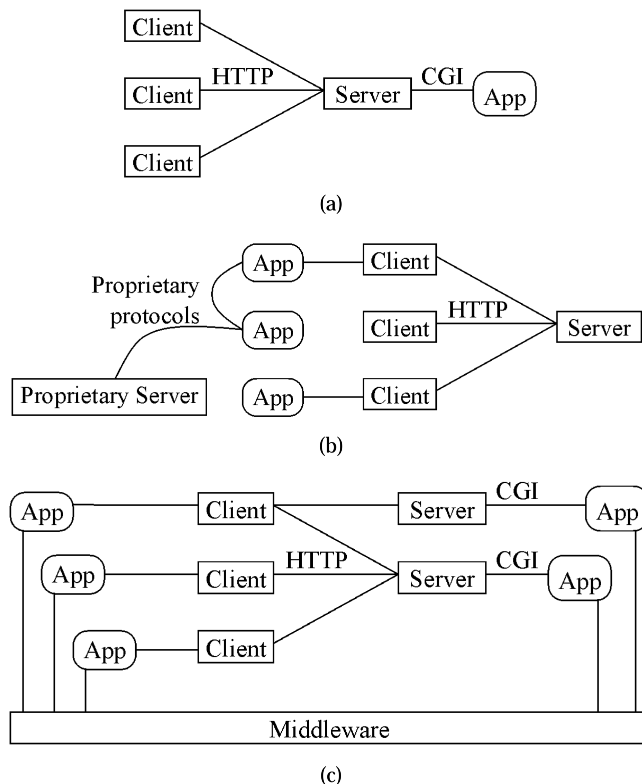

Fig. 1. Activity in Web applications.

Implementing this approach requires language mechanisms like Remote Procedure Calls or Message Passing, which typically provide an Application Programming Interface for posting messages and retrieve responses. The approach we follow instead consists of using a coordination language. The key concept is the use of Linda-like coordination to manage the interaction among agents. This is the topic of the next section.

2.2 Coordination Technology for the Web

The PageSpace software architecture ([4]) is based on the notion of agents that use coordination technology for their interactions. We use the term *agent* reflecting that processing is performed in such an entity. Each user has a *homeagent* that provides the interface to the PageSpace and its agents. Applications are composed by a set of distributed agents therein. We rely on Java as the implementation language for our agents. The main focus of PageSpace is the issue of coordination amongst these distributed, concurrent agents. We explored the use of Linda-like coordination technology to solve that coordination problem.

Three issues are important in a distributed, concurrent application: 1) how agents synchronize their work, 2) how agents communicate, and 3) how agents' activities are started. Among the various approaches to solve these coordination problems, there is a specific line of research called *coordination technology* that is based on the concepts introduced by the language Linda [7].

Linda provides an abstraction for programming concurrent agents and defines a very small set of coordination operations. In a Linda-based system, an ensemble of agents work on a task within a shared environment, called the *tuplespace*. A

tuplespace contains tuples, which are structured containers of information relevant for the application. Many variants of tuplespaces, like distributed, or hierarchically structured ones, have been studied over the past 15 years.

Linda's primitives provide means for agents to manipulate that shared tuplespace, thereby introducing coordination operations. A tuple can be emitted to the tuplespace by an agent performing the `out`-primitive. As an example, `out("amount,"10, a)` emits a tuple with three fields, that contain a string, an integer, and the contents of the program variable `a`. This operation is nonblocking.

Two blocking primitives are provided to retrieve data from the tuplespace: `in` and `rd`. Both take a *template* as argument—for example `in("amount, "?int, ?b)`.

A *matching rule* defined in Linda governs the selection of a tuple from the tuplespace: The template and the tuple must have the same length, the types of the fields must be the same, and the values of constant fields (called *actuals*) have to be identical.

The example template retrieves a tuple that contains the string `amount` as the first field, followed by an integer, followed by a value of the same type as the program variable `b`. The notation `?b` indicates that the retrieved value is to be bound to the variable `b` after retrieval.

The difference between `in` and `rd` is that the former removes the matching tuple, while `rd` leaves it untouched in the tuplespace. Both operations are blocking—while there is no matching tuple found in the tuplespace, they do not return. Linda makes no further guarantees on the selection of matching tuples and waiting operations.

It has been demonstrated [8] that Linda is capable to express all major styles of coordination in parallel programs. `in` is a very powerful operation—it combines synchronization (the operation blocks until a matching tuple it found) with communication (the binding of values to program variables).

All together, Linda's operations form a so-called *coordination language* [9], which, when combined with a sequential programming language, generates a new language for concurrent systems. Such a combination is called *embedding* and can be implemented by changes to the sequential programming language syntax and runtime [10], by preprocessing source code [11], by libraries [12], or can be provided as an extended operating system [13].

Linda-like coordination is attractive for programming distributed applications on the Web because it allows for several unique characteristics not found in other similar technologies, such as Parallel Virtual Machines (PVM [14]):

2.2.1 Uncoupling of Agents

PVM is based on message passing, that means that agents must know each other to communicate. Instead, a tuplespace uncouples the coordinating agents in space and time. An agent can perform an `out` even when a "destination" agent does not yet exist, and can terminate before the `out`-ed tuple is retrieved. The tuplespace abstracts away from locality issues.

2.2.2 Associative Addressing

PVM agents use direct naming. In a Linda program, the template used to retrieve a tuple specifies what kind of tuple is sought, rather than how to find such a tuple. This addressing is more abstract and declarative than specifying a given message from/for a given correspondent.

2.2.3 Separation of Concerns

A coordination model like Linda focuses on the issue of coordination only: a derived coordination language ideally is not influenced by features specific of a host programming language. Interestingly, PVM can be used to implement Linda-like coordination (a project named Glenda did exactly this), however at a high price, in terms of syntactic complexity and lack of semantic optimization.

Linda-like coordination is available for a number of different programming languages and hardware architectures. All implementations usually offer an abstraction of shared tuple space and primitives based on associative addressing. We consider and use here Jada and Laura, two specific Linda-like coordination packages that are described in Section 4.

Coordination can be added to the WWW in at least three different ways as illustrated in Fig. 2.

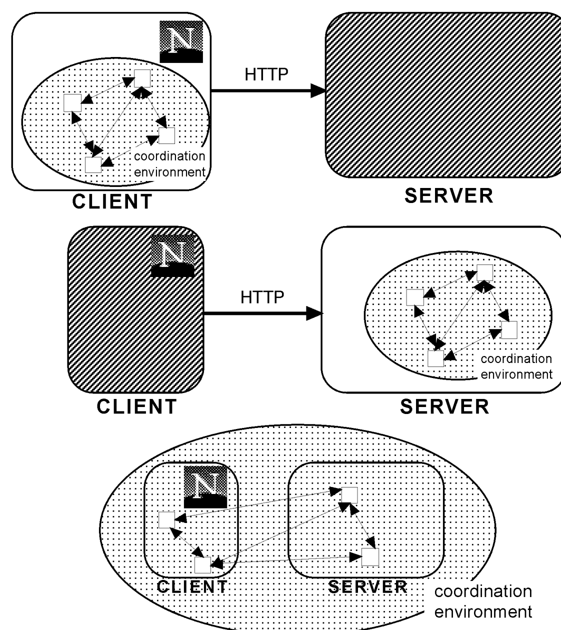

Fig. 2. Three ways to enhance the WWW with Linda-like coordination. Top: client-side coordination. Middle: server side coordination. Bottom: application-wide coordination.

Such a picture illustrates three different ways of using Linda-like coordination to enhance the current WWW softeare architecture. It shows that the introduction of a tuplespace can be useful at all the three layers of a WWW-based distributed application:

2.2.4 Client-Layer

Several clients (or several applets in a client) can coordinate themselves through a shared tuplespace. It is not difficult to coordinate (e.g., synchronize) browsers running on different machines or devices. Alternatively, the browser itself can be composed of several independent modules that are coordinated through a shared tuplespace.

For instance, the WWWinda approach is based on a modular browser The WWWinda approach demonstrates this with a design for a modular browser [15]. To facilitate a tighter integration of browsers and their helper applications, the WWWinda research team designed a flexible, modular browser architecture based on the Linda coordination model. It is composed of several independent tools, each implementing a different part of the Web browser or a helper application. This allows for a highly modular architecture, where new components can be added without modifications to the others. The collaboration among components is implemented with Linda coordination technology: all modules make use of a "software bus" that is a shared tuple space. Current examples include a musical orchestration system (called "distributed karaoke"), in which several independent instruments (possibly even running on different machines) extract from the shared tuplespace the tune to be played note by note. No instrument is aware of how many other instruments are present, and new ones can be added on the fly, even in the middle of a note.

2.2.5 Server-Layer

Different components of a server-side application (or several different applications) can coordinate and communicate through a tuplespace. In this case the HTTP protocol acts as the interface to the modules seen as a whole and is the default access mechanism used to manage the components of the application via a CGI interface. The tuplespace may be used as a connection mechanism for applications running on different machines.

An obvious application consists of distributing the load among several HTTP servers. The WU Linda Toolkit [16] is an example of such an interface to a Linda tuplespace implementation using a WWW browser and an HTTP server. Users can fill out HTML forms with Linda commands that are executed at a shared tuplespace at the server. The main application on show is a disc-load viewer that allows a first glance check of current disk usage of the computers of a cluster. Each computer posts tuples describing its current load. These tuples are then collected and sent to the browser in a graphical HTML display.

2.2.6 Application-Layer

Client/server reference architectures define how an application obtains a service from another. In several situations such architectures put strong constraints on designers, who actually need extended model architectures able to coordinate multiple, independent programs to provide complex services to components which sometimes play the role of clients and sometimes play the role of servers [6].

Coordination architectures [5] are useful to design this kind of applications, because they offer a way to handle coherently and uniformly diverse design issues, like locating resources, supporting interprogram communication, and coordinating the distributed execution.

This paper presents PageSpace, which is an example of a multiagent reference architecture useful to design distributed multiuser WWW applications. In Section 5, we will compare PageSpace with other architectures proposed to build distributed WWW applications.

3 The PageSpace Architecture

PageSpace is a reference software architecture for coordinating applications such that:

1) Applications can be seen as the combination of several independent *agents* whose interaction defines the application behavior.
2) The applications can serve several users independently accessing the shared environment. The users are using the current generation HTML browsers.
3) The applications can all be transparently distributed across several computers or centralized on a single host. The actual settings of each shared workspace should have no influence on the implementation of the applications.
4) Several independent applications that run independently can interact. Coordination and communication may happen not just among different modules of a single, complex application, but may arise naturally from independent applications dealing with the same types of information.
5) The configuration of users, applications, and hosts can change and evolve dynamically with none or minimal disruption of the services of the shared environment. In particular, users are supposed to log in the system using standard HTML browsers on possibly unreliable and/or nonpersistent connections. A user therefore may need or want to change the page currently displayed in the browser, may be subject to network interruptions, or may opt to close a dial-up connection during the run-time of the application, and log back in some time later. These situations should not interrupt the regular functioning of the environment and of the applications, and should gracefully allow the disappearance and reappearance of the users.

In the PageSpace reference architecture, therefore, we distinguish several kinds of *agents*:

1) *User interface agents* are the interfaces of applications. They are manifested as a display in the users browser and are delivered to the client by the other agents of the application according to the requests of the user. Depending on the complexity of the application and the capabilities of the user's browser, there may be different instantiations of user interface agents (in HTML, JavaScript, Java, etc.) that are displayed or executed on the browser. User interface agents are displayed within a general interface framework that provides support for the stable interface elements to manage the interaction with the homeagent.
2) *Homeagents* are a persistent representation (avatar) of users in the PageSpace. Since at any moment users can be either present or absent in the shared workspace, it is necessary to collect, deliver, and possibly act on the messages and requests of the other agents. The homeagent receives all the messages bound to the user, and delivers them orderly to the user on request. Evoluted homeagents can in some circumstances actively perform actions or provide answers on behalf of the user in her absence.

3) *The coordination architecture* is not an agent: it is the operating environment, a shared workspace, where the agents live and communicate. Different coordination architectures may provide different capabilities and, ultimately, a different paradigm for creating the agents of the application. In Section 4 we will mention two different coordination architectures we are using, Jada and Laura.
4) *Application 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 on the coordination architecture, 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 human in any way, and therefore will have no need for a user interface. They will just use and offer services to other agents. Some other application agents, on the other hand, will have to be controlled and monitored by a human. In this case, they will provide on request the user interface agent, in the form of an HTML document, a Java applet, etc., which will be delivered to the requesting user as a normal message and will be displayed or executed on the user's machine.
5) *Gateway agents* provide access to the external world for PageSpace applications. Applications needing to access other coordination environments, 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 environment, and will translate and deliver externally the services requests of the application agents, and will deliver the corresponding response back to the appropriate agent.
6) *Kernel agents* provide sensible 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 movement 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.

We call "agents" the entities present in the PageSpace architecture since they are more than pure objects: The application agents are autonomous and can be active, homeagents work on behalf of the user, etc.

The selected set of agents can be considered a reference architecture since they provide a definite set of components and interactions among them that correspond naturally with the set of requirements seen previously [17].

In Fig. 3 we summarize the PageSpace architecture. 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, a CORBA-based interoperability environment, or some legacy application, like a spreadsheet, or e-mail and news.

The PageSpace environment is maintained by a set of kernel agents on different nodes. Note that application and gateway agents are location independent, the user interface agent is located at the user's machine, the homeagent is at some fixed location, and kernels are present on each participating machine.

In the following, we describe these agents in more detail.

3.1 User Interfaces and Homeagents

The PageSpace and its applications are accessible to the user from any Web browser. This browser may usually be located on a different machine than the actual agents performing the applications. The user activities are not tied to the applications running on the PageSpace. For instance, he/she can display other pages, activate other applications, even disconnect him/herself from the network for some time. Also, the user can move from one browser and machine to others during the lifetime of the applications.

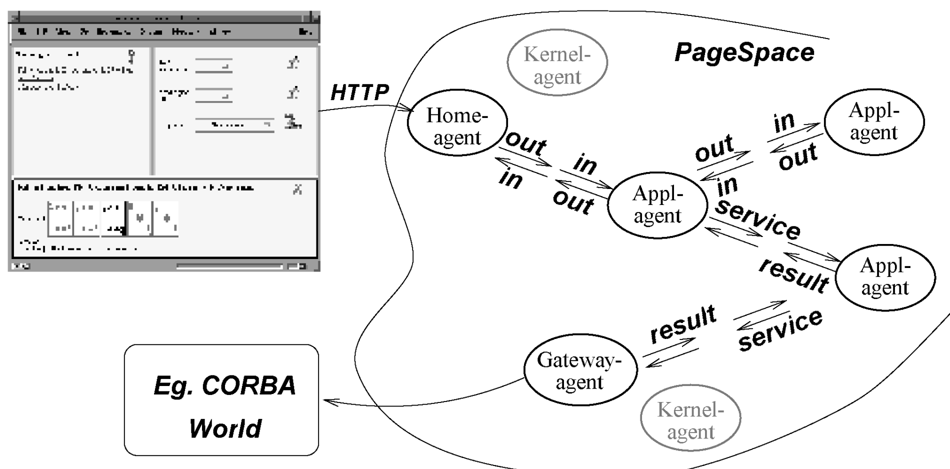

Fig. 3. An application in PageSpace.

Access to the applications of the PageSpace is performed through the user interface agents. User interface agents are delivered by the application agents whenever the user requests to interact with the application. The interface, in the form of a plain HTML document or a Java applet embedded in an HTML document, is delivered to the user's browser through the homeagent.

Given the unpredictable behavior of the user activities, the unpredictable availability of the user's machine, and the unpredictable type of connection that has been established between the user's machine and the rest of the PageSpace, user agents can not be considered fully participants to the shared workspace.

In the simplest situation, the user interface agent is an HTML document (e.g., a form) delivering messages through an HTTP connection to the homeagent, which in turn delivers them to the appropriate application agents of the running computations, and returns back the relevant responses.

More sophisticated user agents are interactive Java applets that may even establish a proper connection to the closest machine hosting an instance of PageSpace. In this case the local computer establishes a local instance of the PageSpace where application agents can perform useful computations. These agents will nonetheless be required to gracefully degrade their participation in the PageSpace according to the user's behavior (e.g., when the user leaves the current HTML page), and to the state of the connection. Furthermore, the user may request the current display to be interrupted and reinstated on a different browser on a different machine.

These reasons imply that as little local state as possible is stored on the user agent, and that it is possible to suspend a running user agent and restore it anew at any moment, with no harm to the running applications.

These characteristics show the potential of PageSpace to support any of the structures for applications on the Web as outlined in Section 2.1:

1) *Server located activity* is trivially supported by having all the application agents running on a remote host, and interacting with the user through the homeagent and a plain HTML page as the interface.
2) *Client located activity* is obtained by activating from a client a local (i.e., in the same host) instance of the PageSpace where application agents can be transferred and run. Communication between the local application agents and their interfaces is, therefore, faster and more direct.
3) *Middleware mediated activity* is obtained distributing several independent application agents somewhere on a LAN. They cooperate and communicate with users' clients (on different hosts as well) through their homeagents.

A homeagent is a persistent representation of the user in the PageSpace. From the active interface, a user can use applications and start agents. However, she does not have to be online while the application is running. Consider as an example a groupware application in which users all around the world participate in some work. It would be unacceptable to force the users to be logged all the time, as their tasks are asynchronous in nature. Thus, a user is free to log in and out. While a user is not connected, her homeagent can still receive messages from one of the applications she joined in.

To the PageSpace, the homeagent looks like an autonomous agent. It has full access to the coordination architecture, and sends and receives messages. For application agents, the homeagent is the default destination of all the messages meant for the human user. That is, only the homeagent "knows" whether the user is currently present or absent from the PageSpace.

The homeagent stores all the incoming messages in a persistent store until the user retrieves them and reacts to them. If there is a bidirectional connection in place between the homeagent and the user interface agent, then the homeagent can deliver the messages to the user as soon as they come in. If the user accesses the PageSpace through an HTTP connection, on the other hand, the application agent queues all messages and delivers them as soon as the user agents opens an HTTP connection to the homeagent. We are exploring mechanisms to give homeagents a limited autonomy in responding automatically to incoming messages whenever possible.

Since the homeagent is the collector of the messages bound to the user agent (and, ultimately, to the user him/herself), the user agents interacts directly only with it. A stable part of the user interface should therefore be assigned to interacting with the homeagent.

In Fig. 4 we show a prototype interface for a Poker application. The interface is divided in three parts: the lower part contains the specific interface for the application currently considered by the user. The top right side contains a form whereby the user, through a plain HTTP connection if necessary, can instruct the homeagent to perform some operations, such as activating or destroying an application, get a list of the latest incoming messages, etc. On the top left side, a list of recent messages is displayed from some of the currently active application agents around the PageSpace.

This list is updated by the homeagent on receiving a new message if a persistent, bidirectional connection is in place, or, if relying on standard HTTP connections, either automatically by the browser at regular intervals through client pull technologies, or manually by the user whenever he/she asks for an immediate connection.

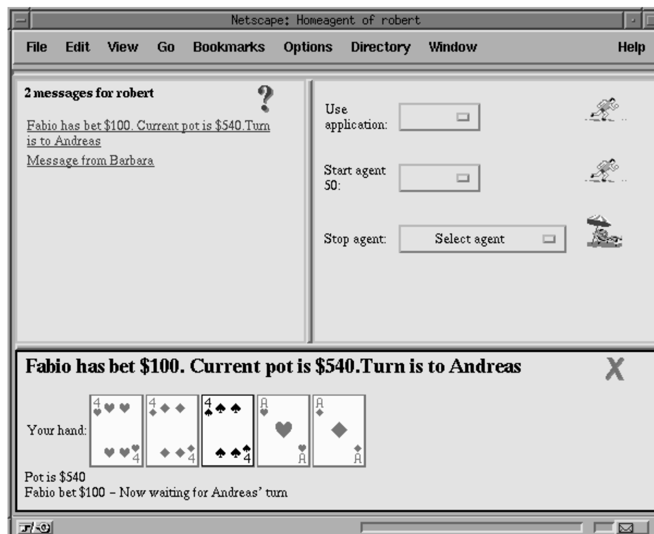

Fig. 4. An interface to PageSpace for the user.

3.2 Applications and Agents

Applications in the PageSpace are composed of agents. In an application in use, three sorts of them are involved:

1) Application agents, which provide the specific user-interface of the application. As described in Section 3.1, we separate the user interface and the application in order to give access to it via a Web browser. Thus, an application agent can define a user interface, for example as an HTML document or a Java applet, which is then passed to the homeagent of a user of the application. From there, it is retrieved and displayed as part of the user interface agent in a browser.
2) Agents that implement the actual application specific functionality. A subset of a distributed application consists of agents that share an application specific context, which is of no use for other agents.
3) Agents that are used by an application, but that offer services to any application. These services are generic in that they can be used in the context of any application.

All agents are started by users within the PageSpace. They remain therein and answer to service requests by other agents until they are withdrawn by their owner. Several kinds of accounting for the use of agents and services can be introduced here.

Agents are programmed using specific classes. With the inheritance mechanisms of Java, a default behavior is provided and a mandatory minimal functionality enforced. The programmer of an application agent focuses on the implementation of the functionality of the service offered.

A designer using PageSpace should take care of the following:

- Users' interaction with the PageSpace;
- Instantiation of user interfaces;
- Management of coordination by offering an API for the interaction with other agents;
- Management of distribution and concurrency;
- Provision of a skeleton functionality for application agents.

The integration of legacy applications and gateways to other coordination environments can be achieved by wrapping and gateway agents. They are similar to application agents in that they offer services to the PageSpace, however, they implement them by interacting with a closed application or via some middleware protocol to other middleware specific object.

A gateway agent that wraps a legacy application offers its services to the PageSpace just like other application agents. However, the implementation of the functionality is embodied in the application being wrapped. The gateway agents *passes* this functionality as services.

A gateway agent which interfaces to another environment is concerned with *mediating* requests from the PageSpace environment to appropriate services found elsewhere and *translating* requests and answers. It may use certain knowledge to ensure semantic correctness of this translation.

4 Implementations of the PageSpace Platform

The PageSpace architecture has currently been implemented in a few prototypes used for demonstration purposes and for the exploration of further concepts. Our prototypes follow the implementation strategy outlined in the previous section. While work remains to be done on the engineering of the architecture, we believe that its main principles can remain unchanged. Some of the prototypes we built are available on line (see the last section for links).

4.1 Basic Coordination Technology in PageSpace: Jada

Jada is a language which extends Java with the Linda coordination language [18]. Jada is based on a set of Java classes to be used to access shared object spaces.

An `ObjectSpace` is an object container offering a set of methods for accessing its contents using a Linda-like coordination model. An object space can be shared among multiple threads since its access is thread-safe, because the accessing methods manage critical sections. To create an object space we write `ObjectSpace my_object_space = new ObjectSpace();`

Following the Linda coordination model, `out` is the method to put an object in the `ObjectSpace`, whereas `in` and `read` are the methods to get an object from the `ObjectSpace`. Actually Jada implements several variants of these basic operations. For instance, to put an object in the object space we write `my_object_space.out(new String("foo"));`

To access the `ObjectSpace` searching for objects an associative matching mechanism is used: a call to the `in` (or `read` method includes an object to be used as a matching pattern. The object returned by the `in` method (if any) is an object found in the the object space that matches the given pattern. The same applies to the `read` method, although only `in` removes the matching object from the object space.

Tuple-matching is based on the concepts of *formal* and *actual* objects: a formal object is an instance of the `Class` class (the meta-class used by Java). Any other object is an actual object. Then the Jada matching rules are:

1) *actual-actual*: two actual items match if:
   - they implement the `JadaItem` interface: the method `matchesItem`, applied to the object in the object space, passing as parameter the other object, returns `true`.
   - they do not implement the `JadaItem` interface: the method `equals`, applied to the object in the object space, passing as parameter the other object, returns `true`.
2) *actual-formal*: a formal item matches any object which is an instance of it.
3) *formal-formal*: two formal items match if they represent the same class.

The matching mechanism in Jada is thus object-oriented; this means that inheritance is applied when checking for two objects to be of the same type: the returned type from an `in` or `read` operation can then be a subclass of the specified object. Note that inheritance is not applied to formal-

formal-matching. So, for example, `object_space.in(new Object().getClass())` collects a nonformal object stored in the object space.

The Jada matching rules are quite simple yet powerful. Since the actual-actual matching is customizable (using either `equals` or `matchesItem` methods) the mechanism is also very flexible: we can, for example, write an Integer wrapper class with ad-hoc matching rules so that an object matches any object whose contents is less that the one given in the `in` or `read` call. However, sometimes is necessary a way to associate values, like an integer with a string that represents the meaning of the integer, such as `"counter."` Thus, like in Linda, we introduce the `Tuple` class, an object container class with an extended matching mechanism.

In Jada a *tuple* is a set of objects (also referred as *items*) and it is implemented via the `jada.Tuple` class. This is an example of Jada tuple:

```
Tuple my_tuple = new Tuple(new Integer (10),
"test");
```

Such a tuple includes two items (we say that its cardinality is two): the first item is an `Integer` object, the second one is a `String` object. We define actual and formal items within a tuple the same way we defined them for Jada.

To use the associative object space access with tuples we can use the tuple matching mechanism: two tuples `a` and `b` match if they have the same cardinality and each item of `a` matches the corresponding item of `b`. The usual Jada mechanism is used to check if the items match.

Thus, the tuple:

```
Tuple a = new Tuple (new Integer (10)," test");
```

matches the tuple:

```
Tuple b = new Tuple (new Integer (10), new
String().getClass());.
```

Note that to exchange a tuple (and generally any kind of object) two threads do not need to perform synchronous `out` and `read` operations (Jada does not need rendezvous communication). In fact, suppose the threads `ta` and `tb` have to exchange a message: `ta` will put a message inside the object space, `tb` will read the message from the object space. If `ta` performs the `out` operation before `tb` performs the `read` operation it does not have to wait for `tb`: It simply continues its execution, and the tuple is now stored into the object space. When `tb` performs the `read` operation it will be able to read it.

Instead, if `tb` performs the `read` operation before `ta` performs the `out` operation, `tb` will be blocked until an object that satisfies the `read` request will become available (i.e., until `ta` performs the `out` operation).

The `in` and `read` methods are blocking. To avoid blocking a thread when a matching object for the `in` and `read` operations is not available we can use the `in_nb` and `read_nb` methods. They access the object space the same way as `in` and `read`, but return `null` if no matching object is available.

A more sophisticated flavor of `in` and `read` that aborts after a time-out is also available. It is also possible to associate a time-out to each object put in the object space: when the time-out is over the object can be garbage-collected and deleted from the object space.

To allow remote access to an object space, the `jada.net.ObjectServer` and `jada.net.ObjectClient` classes are provided. We used a client/server architecture to manage the object spaces; in fact, each object space is a shared remote resource accessed through an object space server.

The object space server is addressed using the IP address of the host it runs on and with its own port number for a socket connection. This way we can run almost as many object space servers as we like in a network, so that applications can independently operate on several, distributed object spaces. `ObjectServer` is a multithreaded server class which translates requests received from the `ObjectClient` class in calls to the methods of the `ObjectSpace` class.

In fact, both `ObjectServer` and `ObjectClient` are based on `ObjectSpace`. `ObjectServer` and `ObjectClient` communicate using sockets. `ObjectServer` uses `ObjectSpace` to perform the requested operations.

The `ObjectClient` class extends `ObjectSpace` changing its internals but keeping its interface and behavior (apart from some new constructor and some new methods). Thus, a `ObjectClient` object is used just like an `ObjectSpace`, except that it provides access to a remote object space which can run in any host of the network. What `ObjectClient` does is to interface with a remote `ObjectServer` object (which holds the real object space) and requests it to perform the `in`, `read`, and `out` operations and (eventually) to return the result.

4.2 Service Interoperability in PageSpace: Laura

Laura is a coordination model based on Linda where the "tuplespace" is enhanced into the concept of a *service-space* which is a collection of *forms*, special tuples shared by all agents. A form can contain a description of a service-offer, a service-request with arguments, or a service-result with results.

In Laura, a service is described as an interface consisting of a set of operation signatures. The signatures describe the types of the operations in terms of their names and their argument- and result-types. It is, therefore, a record of function-types. A form contains a description of this interface-type for service-identification. Putting a service-request form into the service-space triggers the search for a service-offer form so that the interface-type of the offer is in a matching relation to that of the request.

In Laura, no names for interfaces are used to identify services or for the types of data involved in an operation. Instead, a service offered or requested is described solely by an unnamed interface signature consisting of a set of operations signatures. The operation signatures consist of a name and the types of arguments and parameters.

Laura, therefore, emphasizes *what service* is requested, not *which agent* is requested to perform it. A crucial point therefore is the identification of services.

In Laura interfaces are notated in a service type language. In [19] we formally defined a type-system which is used in the definition of the semantics of such interface definitions. Such a type system includes rules for subtyping and this subtyping is the key for Laura's identification of services: given the interface descriptions in forms, a service offer matches a service request, if the type of the offered interface is a subtype of the requested one.

Subtyping in Laura is defined so that a type A is a subtype of B if all values of type A can be substituted when a value of type B is requested; the "values" we type are services. Such a subtyping enables us to use a service of type A if a service of type B is requested.

Specific of Laura's type system is that it deviates from approaches to the management of types in open systems in three ways. First, it abolishes global names for interfaces for services and relies on matching of interface types only. Second, it ignores names of structured types. Finally, it uses syntactic equivalence only for the names of operations appearing in an interface.

A service is the result of an interaction between a service-provider and a service-user. In Laura, two operations coordinate this interaction for the service-provider, `serve` and `result`.

An agent willing to offer a service to other agents puts a serve-form into the service-space. It does so by executing `serve`, which takes as parameters the type of the service offered and a list of binding rules that define to which program variables arguments for the service should be bound.

When a `serve` is executed, a serve-form is built from the arguments. Then, the service-space is scanned for a service-request form whose service-type matches the offered service by being a supertype. The provided arguments are copied to the serve-form and finally bound to program variables. `serve` blocks as long as no matching request-form is found.

After performing the requested service, the service-provider uses `result` to deliver a result-form to the service-space. A result-form is built which consists of the service-interface and—depending on `operation`—a list of result values according to the binding list. The agent is responsible to store the results of the service properly in those variables. The operation is performed immediately and the form is put into the service-space. An agent offering services usually operates in a loop consisting of the sequence `serve`-*perform the service*-`result`.

An agent that wants to use a service has to execute Laura's third and last operation, `service`. Its arguments are the service-type requested, the operation requested, arguments for the operation, and a binding-list.

When executing `service`, two forms are involved: a service-put form and a service-get form. The first is constructed from the service-interface and the arguments and then inserted to the service-space. If another agent performs a `serve`-operation and the service-put- and serve-forms match, the arguments are copied as described above and the service-provider starts processing the requested operation.

The service-get form is constructed from the service interface and the binding list for the results. Then, a matching result-form is sought in the service-space and—when available—the results are copied and bound to the program variables. When the request-form is entered to the service-space, it is matched with some serve-form. When the result-form is retrieved, the results are bound to program variables of the requesting agent.

The interaction of agents coordinating services with Laura consists either of putting a request for a service to the service-space, finding a matching offer form and copying of arguments or of trying to get the results of a service, by finding a matching result-form and copying of the results. This interaction is uncoupled, as service-provider and service-user remain completely anonymous to each other.

Laura provides the basic means to implement the service exchange among application agents. We adapt the paradigm of exchanging form within the Laura API. With the reflection API in Java 1.1, we could well change from our service description language to Java interfaces in forms.

4.3 The PageSpace Kernel Agents

Every implementation of the PageSpace architecture will have to provide a way for agents to be able to be started, blocked, moved, and generally managed within the selected coordination environment. We have designed a specific class of agents to deal with these tasks, the *kernel agents*. Our implementation runs on a Java virtual machine and manages multiple threads. It requires that one kernel agent runs on each machine participating to the PageSpace. The kernel agents provides several services:

4.3.1 Provision of Access to the PageSpace

A user accesses PageSpace via her homeagent that is contacted by HTTP from a browser. Thus, a Web server has to be co-located with a kernel. As there are several implementations of Web servers in Java—like Jigsaw from the W3C—the HTTP server could be integrated as a thread of the kernel, thus avoiding the CGI mechanism to pass information to homeagents.

4.3.2 Management of Homeagents

Homeagents are in fact implemented as a single object within a kernel. They are parameterized with the identification of a PageSpace user. Thus, after passing a login form in which a PageSpace user name and password is entered, each user receives the same user interface agent components, but each one is based on a different message queue stored persistently in a database on the kernel.

Besides interacting with messages, the user can use applications, and start application agents from its homeagent. Both result in the execution of a thread within the homeagent object. To use an application, that thread issues the appropriate coordination operation, waits for the results, stores it in the database and terminates. To start an agent, a method of the kernel object is invoked which starts an application agent.

4.3.3 Management of Application Agents

Each application agent is executed as a thread. This is reasonable, as we can make use of the native interaction mechanisms within one Java virtual machine for threads, and to not have to execute multiple virtual machines on the node that participates in PageSpace.

The kernel agent manages program exceptions and monitors the operation of the agent threads. Thus, it establishes an operating environment for coordinated application agents within a Java virtual machine. We inherit all aspects of thread management from Java, and add the "multiagent" interaction mechanisms by coordination technology.

4.3.4 Implementation of the Coordination Operations

Several flavors of coordination environments can be used in PageSpace. All of these have in common that they are

centered around the use of a shared space of element of some kind, and that some matching rule guides the coordination primitives.

Thus, each kernel contains instances of a generic component, the *repository*. A repository is a collection of elements of some type. Each repository implements the specific operations of a coordination language with a specific matching routine, thus it may be optimized, but it is based on the management of a pool of elements of some type. Within the kernel architecture, multiple repositories can be integrated to the internal control and data streams.

4.3.5 Implementation of the Distribution Architecture

Each kernel has an interface to other kernel agents, through which the distribution protocol is spoken. We foresee the possibility of different distribution architectures integrated via a set of interconnected sub-PageSpaces.

Fig. 5 shows an excerpt of the logical structure of our implementation of the kernel agents. All the objects that run in threads are connected by sets of streams for both data exchange and agent management.

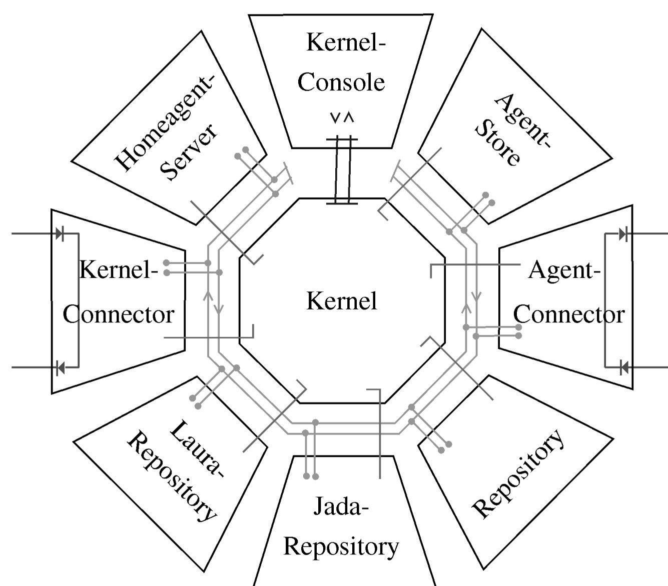

Fig. 5. An outline of the logical structure of a kernel agent.

A kernel agent includes: a kernel console, to control its activity; an agent store, which stores application agents programs; a homeagent server, which stores homeagents; some agent connectors, to let agents interoperate with agents in other kernels; some repositories, which persistently implement some flavor of coordination tuplespace; some kernel connectors, to implement distributed PageSpaces.

In middleware platforms, the issues of discovering, naming, and accessing services are central. In PageSpace we do not enforce a (distributed) registry of existing agents, but take a different approach similar to the way of accessing pages in the Web.

Web pages are named and referenced by URLs and accessed by contacting the corresponding Web server. Similarly, information about agents is offered via HTTP by the kernel agent at a given site. This meta information consists of formal data, like the interfaces of services offered by an agent, and nonformal parts, like a natural language description of the agent.

An agent can thus be accessed by retrieving this meta information via a URL and using it appropriately in coordination operations. The meta information will be generated dynamically by the kernel agent.

The discovery of information in Web pages is currently mediated by catalog services and search engines. In analogy, we foresee catalogs of links to agent meta data and search services on them. Users collect links on Web pages of interest in personal catalogs as bookmarks. By the same analogy, homeagents can offer the user such personal lists of interesting agents and applications.

4.4 Distributed PageSpaces

We have extended our prototype kernel implementation to handle distributed PageSpaces.

Our implementation of the kernel agents manage and coordinate agents on one machine. For distributed applications, these kernels have to have a distribution architecture and a communication protocol. A special concern with such a protocol is scalability – the ability to provide efficient coordination for a platform involving a large number of machines.

The approach of establishing a shared repository of information can lead to scalability problems due to the amount of overhead for replication. We can take a flexible approach to structuring the system to overcome these problems.

We follow the approach of the Internet to scalability: the machines that participate in the PageSpace are organized in a loose federation. Locally connected machines follow a replication schema in a logical sub-PageSpace and one machine is defined as the gateway to other sub-PageSpaces. Thereby, we imitate the interconnected subnets of the Internet.

The specific organization of kernels within one sub-PageSpace is a local decision. Known architectures for distributed implementation of Linda-like systems include full replication of a repository to all nodes, no replication with a single, centralized repository, or a partial replication as in [20]. As long as there is one defined node that follows a gateway protocol to other sub-PageSpaces, our architecture supports all of them. In fact, the current Jada implementation uses a centralized or fully replicated repository, whereas Laura implements a partial replication scheme.

For a gateway, a “routing-table” exists that instructs the gateway as to which other sub-PageSpaces should be forwarded the requests for matching elements. Thus, the distribution structure can be configured statically or dynamically.

This configuration will be based on the structure and behavior of the agents within a sub-PageSpace, and supports them in their coordination requirements. All the flavors of coordination employed in PageSpace give way to several intelligent optimizations that are yet to be evaluated.

4.5 Options for Fault Tolerance and Mobility in PageSpace

The PageSpace architecture has several features that are yet to be explored. We discuss two of them, namely fault tolerance and mobility. We show how these features have been introduced to the platform, and how they are enabled by the design of the platform.

4.5.1 Fault Tolerance

What happens when agents in our architecture fail? Failures of the user interface agents—because of a crashing browser, or a fault in the users machine—do not affect the PageSpace at all. The failure of a homeagent does not introduce problems, insofar as the queue of messages for a user is kept persistent.

Homeagents, application, and gateway agents are managed by the coordination kernel. This means that the kernel can keep a log of their external interactions and request state information that is stored persistently. A kernel thus can monitor the managed agents, and restart them in case they crashed with a given state. We foresee that any managed agent can provide a method that transfers state information to the kernel agent.

In the case of an kernel failure, the kernel and all managed objects are lost. The log of external interactions can be used to reestablish the repositories after restart; the managed agents can be restarted accordingly.

4.5.2 Mobile PageSpace Agents

As application agents interact transparently with respect to their location, they are candidates to support mobility of agents within PageSpace. In order to do so, they have to be able to pass their internal state to a kernel, which can start it.

Application agents may want to be moved because they detect that they interact with each other and try to make the coordination more efficient by "meeting" at a specific location. They can be asked to move by an authoritative kernel, because of a specific policy applies to their current location (eg. workload management, or dedicated machines). It has yet to be evaluated what protocols are most efficient to perform such operations, and what strategies for mobility should be followed.

4.6 Using PageSpace to Design an Application

The PageSpace has been introduced for interactive multiuser applications, so usually we start designing a PageSpace application from defining which user roles need a specific user interface agent.

For instance, we realized an auction bidding system within the PageSpace project as a case study in electronic commerce. The auction system has three types of users: the *Auctioneer*, who sells items to the highest offer, the *Participants*, who buy items sold during the auction, and the *Observers*, passive audience to the auction. The auctioneer and the participants communicate during the bidding, whereas the observers simply follow the bidding.

The activities of the users can be divided in the following phases:

Phase 0. The auctioneer provides information about the auction using messages to newsgroups and mailing lists, and setting up a WWW site. The auctioneer provides for an information package explaining the modalities for the users interested in the auction. Users can either simply observe the auction, or actively participate to the bidding. In the first case, they are informed of the bids, but they are not allowed to place a bid and buy an item. In the second case, they can actively participate to the bidding, sending bids for an item.

The auctioneer then starts an application agent building the environment for the auction. All participants and observers activate their own application agents to handle their bids. To do so, they ask their homeagent to start up a new specialized application agent. All messages relevant to the auction are exchanged among these agents. If a user is not online at bidding time, he can be represented by an autonomous agent, which may follow some bidding policy that does not require a direct user intervention.

Phase 1. The auctioneer puts an item up for auction and starts a timeout for receiving bids. He notifies to all connected users the new item and its base price. As soon as the new item is put on sale, the auction agent notifies all participants and observers.

Phase 2. As soon as the information about the new auction item is made available to the participants, they can submit the auctioneer a new bid, or request to unsubscribe from the current item, i.e. by stopping to receive updates for the current item and waiting for the next one.

*Phase 3.*The auctioneer accepts any valid bid (i.e., larger than the current bid) from registered participants, and resets the timeout value. Every new bid causes a broadcast of the new value to all participants and observers. When no new valid offer is received within the time out period, the auction stops and the item is considered sold. The auctioneer notifies all users that the sale is closed, and starts a new auction for the next item, resetting the list of participants.

The user interface agent for the auction system is shown in Fig. 6.

It is divided into three frames containing a number of applets. The top left frame displays the item currently offered. The top right frame allows users to place bids, and on the bottom the currently valid bid is displayed. Every time a new valid bid is received by the auctioneer, all bottom frames are updated with the new value. After a time out, an animation of a mallet hitting a surface is displayed to signify a final sale.

Fig. 7 shows the applets which compose a user agent: they coordinate via Jada both locally and remotely.

The agent `bid` is interactive: it allows a customer to place a bid. This applet is always active, waiting for user input.

The customer has to choose an amount to increment the current bid (displayed by applet `display`). The customer must insert in `TextField` the username; the amount is chosen by the (`Choice`) button (in Fig. 6 this shows `$550`) finally, the user has to press another button to send the offer.

Agent `next_item` is a button which allows to skip the current item on auction to another one.

Agent `cartoon` animates a gavel representing that the current auction is still in progress, or it is finished (the gavel falls). The animation is based on a sequence of five images.

Agent `display` shows the current bid and its owner. This agent is especially critical, because it must show the same information to all bidders.

The most critical agent is the `auctioneer` which is the logic coordinator of the whole system and can run on any host: it decides when an offer is valid; it tells agent `display` about the current bid; it sells an item, etc. This agent is not displayed by an applet. The tuple space has to run on a server machine running also the auctioneer for security constrains imposed by Java. Other agents have no constrains at all.

Fig. 6. The interface agent of the auction system.

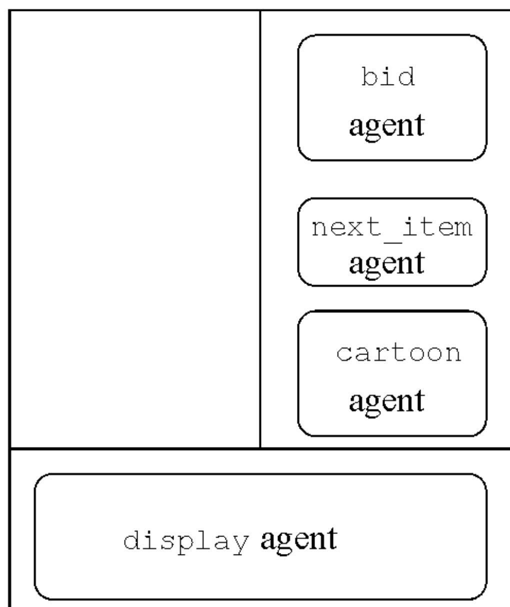

Fig. 7. Applets in useragents.

This is a simple yet realistic example of using the PageSpace reference architecture in a context of electronic commerce. In the application we have developed, we allow customers to participate offline to the auction, and we interoperate via a gateway agent with a database containing images and descriptions of auction items. The behavior of agents representing customers is defined by rules in a manner very similar to that described in [2].

As another case study we developed a distributed WWW-based card game. The game itself is a variant of the well-known *Hearts* card game. The components of this interactive distributed application are:

- the players,
- the table server,
- the hearts servers, and
- the chat servers.

The table server manages the arrangement for the tables: players can join/leave a table sending a request to the table server. When a table has four players the game starts. Once the game is started the players have to talk with a hearts server, created on the fly by the table server, in order to compete in the game. While the players are arranging the table or playing they can exchange text messages using the chat server.

5 Related Approaches

Several research and commercial efforts are currently concerned with adding various functional features to the Web. Being based on a client-server software architecture, the WWW can be extended in three different parts: either the server, or the client, or the communication protocol, or a combination of them. Here we present a few academic and industrial projects that show examples of these extensions.

Middleware standards such as CORBA and DCE are the results of the effort of industry committees consecrated to making commercial applications interoperate as smoothly and as painlessly as possible for the application designer, the system developers, and for the final users. These platforms provide standard ways to define, locate and request computational services from participating applications, both locally and remotely. Middleware solutions may help the WWW define a standard and unique way to access and execute remote services, letting it potentially become the standard interface to all networked services available now and in the future.

For instance, ANSA is pursuing the integration of WWW technologies and CORBA-related standards [21]. Their approach is two-sided: on the one hand, they are creating a standard set of CGI gateways to allow bidirectional interaction between a CORBA based environment and HTTP tools (CORBA clients accessing to HTTP resources and HTTP software accessing CORBA distributed objects). On the other hand, they are building a set of WWW tools to integrate and replace HTTP: a server that can provide services using both HTTP and IIOP (the Internet Inter-ORB Protocol providing the connection layer to all CORBA 2.0 compliant platforms) and an Arena-based browser using IIOP as its connection mechanism.

On the other hand, Marco is a CGI gateway to OSF's DCE servers [22] developed as a demonstration of a proposed general architecture for integrating generic middleware components into the WWW through CGI. Two modules are identified: a "type manager," which knows about data contained in the middleware services connected, and a "trader," which knows about the details of service instances and interface descriptions of the services connected. A general interaction protocol is defined, allowing clients to identify the service through a two-step request, receive an HTML form document suited to the kind of service requested, and perform the most efficient request.

These two projects are examples for server-centered architectures for Web-applications. They provide only a Web interface to an otherwise not Web-enabled infrastructure such as CORBA and DCE. In contrast, PageSpace is not a mere gateway, it is a reference architecture integrated with the Web and allowing for applications that are in part distributed to the users client.

Two recent developments enable applets to become parts of truly distributed applications. With Netscape Communicator 4, a CORBA interface is available for applets with the inclusion of the Visigenic ORB and the respective Java interface. The revision 1.1 of Java introduced a standardized way of communication amongst distributed Java objects, the Remote Method Invocation API.

Similar to PageSpace, both enable truly distributed applications, either in an open CORBA world, or within the Java paradigm. In contrast to PageSpace, both rely on client-server technology. PageSpace is different in the characteristics of the coordination technology applied. The main specific difference is that any remote invocation in CORBA or Java RMI is directed towards a specific, existing object, whereas PageSpace employs undirected coordination which is decoupled in space and time.

Jada applets are similar to Oblets, or "distributed active objects" [23]. An *oblet* is a program written in Obliq and executed on a Obliq-aware browser. Each oblet can use high-level primitives to communicate with other oblets running on possibly remote browsers. Instead of Obliq, Jada uses original, pure Java enhanced with simple coordination primitives.

We know of two projects directly based on Linda-like coordination: Bauhaus and JavaSpaces.

The Bauhaus "Turingware Web" [24] designed in Yale University, the homeland of Linda, is similar at least in spirit to the WU Linda toolkit. The main idea consists of using a standard browser to access a Bauhaus server. Bauhaus is a coordination language based on nested multiple tuplespaces which can be used in this case for both controlling the hierarchical structure of the pages of a web site, and for associating agents and their activities to the pages themselves. For instance, one attribute of a page could be the list of users "acting" in such a page, who are displayed by a graphic icon and can interact using some ad-hoc cooperation services.

Sun has announced but at moment not yet released JavaSpaces, a Java library which uses Linda-like coordination for supporting distributed applications [25]. It focuses on transactional Linda operations and is intended as a platform to provide distributed persistence for exchangeable objects.

Both systems are within the same line of research as PageSpace, combining coordination technology and the Web. However, both focus on single issues. The Bauhaus approach is possibly less general, insofar as it redefines the basic WWW software architecture. Instead, JavaSpaces is advertised as a layer added on top of Java allowing for simple programming of applications which require transactional operations. In contrast, PageSpace is a reference architecture based on coordination technology which defines a multiagent WWW infrastructure for coordination-based applications.

6 Conclusion

The PageSpace integrates three basic building blocks:

1) *Web technology*, which provides a uniform communication and presentation platform: standard (Java-enabled) browsers are used to access the PageSpace.
2) *Java technology*, which provides a uniform host language. PageSpace enhances Java with a high level coordination support for distributed agents.
3) *Linda-like coordination technology*, which provides a simple tool to describe and control activities of asynchronous agents.

We remark that the idea of coordination has been subject to a large variety of research projects, focusing on parallel or distributed coordination architectures, on theoretical foundations of coordination languages, and on a number of implementation oriented research efforts concerning the embedding of coordination models into practical programming platforms. We have shown that combining Linda-like coordination with Java it is possible to build a flexible platform to support open distributed applications. The necessary mechanisms turn out to be simple and require no extensions to the building blocks used.

More information on 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 No. 20179. The authors thank the anonymous referees for their comments.

References

[1] E. Ly, "Distributed Java Applets for Project Management on the Web," *IEEE Internet Computing*, vol. 1, no. 3, pp. 21–27, May/June 1997.

[2] J.M. Andreoli, F. Pacull, and R. Pareschi, "XPect: A Framework for Electronic Commerce," *IEEE Internet Computing*, vol. 1, no. 4, pp. 40–48, July/Aug. 1997.

[3] E. Evans and D. Rogers, "Using Java Applets and CORBA for Multi-User Distributed Applications," *IEEE Internet Computing*, vol. 1, no. 3, pp. 43–55, May/June 1997.

[4] P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali, "PageSpace: An Architecture to Coordinate Distributed Applications on the Web," *Computer Networks and ISDN Systems*, vol. 28, nos. 7-11, pp. 941–952, 1996.

[5] P. Ciancarini, "Coordination Models and Languages as Software Integrators," *ACM Computer Surveys*, vol. 28, no. 2, pp. 300–302, 1996.

[6] R. Adler, "Distributed Coordination Models for Client/Server Computing," *Computer*, vol. 28, no. 4, pp. 14–22, Apr. 1995.

[7] N. Carriero and D. Gelernter, "Linda in Context," *Comm. ACM*, vol. 32, no. 4, pp. 444–458, Apr. 1989.

[8] N. Carriero and D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," *ACM Computer Surveys*, vol. 21, no. 3, pp. 323–357, 1989.

[9] N. Carriero and D. Gelernter, "Coordination Languages and Their Significance," *Comm. ACM*, vol. 35, no. 2, pp. 97–107, Feb. 1992.

[10] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnaswamy, "Matching Language and Hardware for Parallel Computation in the Linda Machine," *IEEE Trans. Computers*, vol. 37, no. 8, pp. 921–929, Aug. 1988.

[11] N. Carriero and D. Gelernter, "Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler," *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua, eds., pp. 114–125. Cambridge, Mass.: MIT Press, 1990.

[12] Scientific Computing Associates, New Haven, Conn., *Paradise 4. Reference Manual*, 1996.
[13] W. Leler, "Linda Meets Unix," *Computer*, vol. 23, no. 2, pp. 43–55, Feb. 1990.
[14] G. Geist and V. Sunderam, "Network-Based Concurrent Computing on the PVM System," *Concurrency—Practice and Experience*, vol. 4, no. 4, pp. 293–311, June 1992.
[15] Y. Gutfreund, J. Nicol, R. Sasnett, and V. Phuah, "WWWinda: An Orchestration Service for WWW Browsers and Accessories," *Proc. Second Int'l World Wide Web Conf.*, Chicago, Il., Dec. 1994.
[16] W. Schoenfeldinger, "WWW Meets Linda: Linda for Global WWW-Based Transaction," *World Wide Web J.*, vol. 1, no. 1, pp. 259–276, Dec. 1995.
[17] M. Shaw and D. Garlan, *Software Architecture. Perspectives on An Emerging Discipline.* Prentice Hall, 1996.
[18] P. Ciancarini and D. Rossi, "Jada: Coordination and Communication for Java Agents," *Mobile Object Systems: Towards the Programmable Internet*, J. Vitek and C. Tschudin, eds., *Lecture Notes in Computer Science 1,222,* pp. 213–228. Berlin: Springer-Verlag, 1997.
[19] R. Tolksdorf, *Coordination in Open Distributed Systems.* Number Reihe 10, 362 in VDI Fortschrittsberichte. VDI Verlag, 1995, ISBN 3-18-336210-4.
[20] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM Trans. Computer Systems*, vol. 4, no. 2, pp. 110–129, May 1986.
[21] O. Rees, N. Edwards, M. Madsen, M. Beasley, and A. McClenaghan, "A Web of Distributed Objects," *World Wide Web J.*, vol. 1, no. 1, pp. 75–88, 1995.
[22] A. Beitz et al., "Integrating WWW and Middleware," *Proc. First Australian World Wide Web Conf.*, R. Debreceny and A. Ellis, eds., Lismore, NSW: Norsearch Publishing, 1995.
[23] M. Brown and M. Najork, "Distributed Active Objects," *Computer Networks and ISDN Systems*, vol. 28, nos. 7–11, pp. 1,037–1,052, 1996.
[24] N. Carriero, D. Gelernter, and S. Hupfer, "Collaborative Applications Experience with the Bauhaus Coordination Language," *Proc. HICSS30, Sw Track*, Hawaii, pp. 310–319, IEEE CS Press, 1997.
[25] J. Waldo et al., "Javaspace Specification—Revision 0.4," technical report, Sun Microsystems, JavaSoft Lab., June 1997.

Paolo Ciancarini received the PhD degree in computer science from the University of Pisa, Italy, in 1988. Dr. Ciancarini is an associate professor of computer science at the University of Bologna, Italy. His research interests include coordination languages and systems, programming systems based on distributed objects, and formal methods in software engineering. He has been a member of ESPRIT BRA Project COORDINATION on Coordination models and languages; is a coproponent of the PageSpace Open LTR project; and is a member of the ESPRIT Working Group Coordina "From Coordination Models to Applications." Dr. Ciancarini is a member of the IEEE Computer Society.

Robert Tolksdorf received his Dr-Ing degree in computer science from the Technical University Berlin in 1995. He is an assistant professor at the study group formal models, logic, and programming in the Department for Computer Science at the TU Berlin. His research interests include coordination languages, open distributed systems, and Web-technology. He is one of the main proponents and is responsible for coordinating the TU Berlin site of the ESPRIT Open LTR project 20179 PageSpace on coordination in distributed WWW applications. He is a member of the ESPRIT Working Group Coordina "From Coordination Models to Applications." He is a member of the IEEE Computer Society.

Fabio Vitali received the PhD degree in computer science and law from the University of Bologna in 1994. He is a research associate of computer science at the University of Bologna. His interests include user interface design, hypertext models, markup languages, versioning systems, and coordination languages.

Davide Rossi is a PhD candidate in computer science at the University of Bologna. His interests include object-oriented programming, operating systems design, coordination languages, mobile agents, and compression algorithms for graphic formats. He was involved in the design on one of the earlier packages for JPEG compression.

Andreas Knoche worked in the PageSpace Open LTR project at the Technical University Berlin as a research assistant. He now is with the KIT-ZVLB project at TU Berlin which builds a VRML-based information system. His research interests include coordination languages, open distributed systems, and Web-technology.