A universal information appliance

by K. F. Eustice
T. J. Lehman
A. Morales
M. C. Munson
S. Edlund
M. Guillen

The consumer's view of a universal information appliance (UIA) is a personal device, such as a PDA (personal digital assistant) or a wearable computer that can interact with any application, access any information store, or remotely operate any electronic device. The technologist's view of the UIA is a portable computer, communicating over a bi-directional wireless link to an elaborate software system through which all programs, information stores, and electronic devices can export their interfaces to the UIA. Using an exported interface, the UIA can interoperate with the exporting entity, whether a home security system, a video cassette recorder, corporate application, or an automobile navigation system. Furthermore, interfaces presented by the UIA can be tailored to the user's context, such as the user's preferences, behavior, and current surroundings. The UIA programming model supports dynamic interface style and content triggered on activity detected from the user's real-world and software context. In this paper we describe the design and first implementation of a UIA, a PDA that, through a wireless link, can interact with any program, access any database, or direct most electronic devices through a remote interface. The UIA model uses IBM's TSpaces software package as the interface delivery mechanism and resource database, and as the network communication glue. TSpaces supports communication between the UIA and any peer over a dual-mode wireless link. Using a popular application example, we present a generalized architecture in which the UIA is the mobile user's software portal for interoperating with any peer: another UIA, a common network service, a legacy application, or an electronic device.

People are accustomed to static interfaces. The refrigerator, television, coffeemaker, and thermostat show the same appearance to us day after day. We would be quite surprised if those interfaces changed without warning. Perhaps because of our experience with the physical world, we tend to design our software with the same static model. Moreover, the tendency is reinforced by typical programming environments. Interfaces that vary dynamically (during one run-time instance) are not common, and context-sensitive interfaces (interfaces that change according to one's experience, abilities, or current surroundings) are quite rare.

However, the electronic world does not need to share the physical limitations of coffeemakers and toaster ovens. A single device with a flexible interface could represent an unbounded number of electronic devices and software systems. What if a mechanism were to exist that promoted such dynamic personalizable interfaces? What if each person could have his or her own personalized interface to an ATM (automatic teller machine), a television and a videocassette recorder, a personal stereo system, or the control panel of a home security system? The goal of the universal information appliance (UIA) effort[1] is to create an environment in which a single device can serve as a person's portal into the digital and electronic domain. The UIA effort is composed of several parts that can be broadly categorized into three areas:

1. A user's interface to the digital domain—The user's interface is presented by a wearable computer, which the user presumably carries whenever he or she desires to be part of the electronic network. For their choice of wearable hardware, users will select from many different user devices. Furthermore, it is assumed that what is the standard wearable computer today will likely be replaced by new hardware and operating system implementations in the future. In place of a platform-specific user-interface (UI) rendering, the UIA effort focuses on building virtual machinery capable of rendering user interfaces on any device by translating a common expression for user interfaces into a platform-specific implementation.

2. A wireless infrastructure, keeping the UIA connected to the network—To access the digital domain in any user situation, the UIA requires a link to the network that is independent of physical location. The UIA effort includes a wireless infrastructure designed to support both continuous and intermittent modes of operation. The wireless link provides high-speed, continuous communication for use in a fixed or nearly fixed location, such as an office, home, or store. The link also provides good capacity for intermittent connectivity at longer distances, such as when a user is traveling in a campus, downtown area, or airport terminal.

3. Communication middleware connecting the UIA to services and information—The UIA provides access to whatever service or device interface the user requires in a particular situation—this is its major advantage. The flexible functionality the user perceives is achieved through network middleware that allows the UIA to interact with services in the network as if the UIA were designed to be a client of the particular service. The communication middleware provides a uniform mechanism for accessing devices or services across heterogeneous platforms that is open to application and language-specific data types and allows the UIA both to trigger and to be triggered by events in other entities.

In this paper, we discuss our approach to these three areas of design for the UIA. In the next section, we describe the overall vision of the UIA and the niche it fills in the consumer electronics space. In the subsequent section, we describe the challenges we encountered in designing the UIA. Thereafter we describe our first implementation of a UIA system, detailing the hand-held consumer device and the network middleware. In the fifth section, we describe one of the popular applications that we envision for the UIA. We then go on to discuss our current research in enabling the UIA to universally interoperate with network services. In the seventh section, we present related research and draw comparisons with the UIA approach. Finally, we conclude the paper and discuss some future work.

The vision

Although we live in a physical world, in our daily lives we are called upon to interact with both physical and electronic systems.[2] The physical systems have human interfaces for their use: doors have doorknobs, drawers have handles, sinks have faucets. Similarly, elements of the electronic world present physical interfaces (e.g., the knobs on the stereo system, the buttons on the television, and the keyboard and mouse on the computer) that trigger events or indicate events in the electronic universe. In physical systems, the "better" interfaces—those that make use easier—are intuitive; the user-interface functions directly relate to what the device actually does. For example, the interface to turn on water (usually) involves the manipulation of a valve that releases the flow, such as turning a knob or raising a lever. However, in electronic systems, there is a disconnection between the user interface and the actual function of the electronic entity. How does one decide the "best" interface for repartitioning the resources of a Web farm, redirecting a particular query to a different information source, or reprioritizing a set of system processes? Creating an interface for a new electronic function that is intuitive to everyone is arguably difficult, if not impossible. Consequently, most physical user interfaces to electronic systems are tailored to the greatest *common denominator* of human experience, rather than a *multiplicity* of human experience, and few interfaces, if any, are dynamically tailored to an individual person's needs.

Most people have become resigned to this reality, and, as a result, there has been little hope for a viable solution to the general interface problem. Furthermore, as the number of electronic systems with which we interact increases, we are presented with (and increasingly overwhelmed by) a growing number of untailored, incompatible, and inconsistent interfaces. We develop large collections of remote controls, personal data assistants, hand-held computers, notebook computers, and countless other electronic "interfaces." This multiplicity has led to the devel-

opment of "universal remote controls"[3] and other such devices[4] to interact with limited combinations of electronic devices, to reduce the "remote bloat." The universal information appliance transcends this notion by extending the concept of remote interaction beyond direct control of televisions and video cassette recorders (VCRs) to dynamic interaction with all electronic entities and digital information sources in one's environment. Furthermore, the UIA model tailors that interaction to the specific user's context.

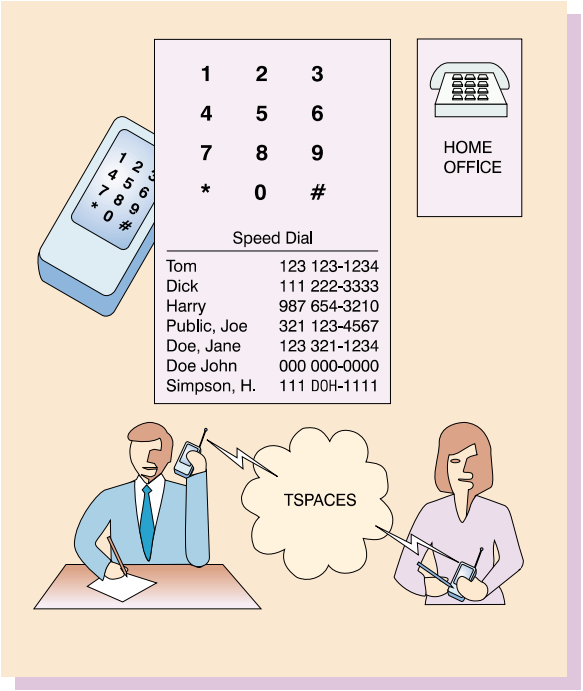
The universal information appliance is intended to be the average person's device for hosting tailored interfaces to the entire electronic universe—interfaces to virtually any device or software program (Figure 1). Figure 1 shows two possible variations of a UIA interface in a telephone-type application. The user on the left calls a variety of numbers, sometimes calling the same number several times in a day. This user has customized the interface on the UIA to display a keypad for dialing new numbers, as well as a list of most recently dialed numbers. The user also prefers a list of most frequently dialed numbers. The user on the right has opted for a simple interface with buttons to call one of two numbers. This user often receives calls on the device, but only makes calls home or to the office. Either user can decide to make further changes to the interface whenever necessary without losing the core functionality (interfacing to a network telephone service). Although the two UIA interfaces have a completely different "look and feel" (which can be changed dynamically), both are suitable clients for the same telephone service.

However, the UIA does more than just display a set of interfaces. It is a companion that stores the user's profile information, performs local computation, maintains the soft state for the user's current applications, and holds a cache of knowledge to which the user wants immediate access.

Consider the following scenario:

"BLARP, BLARP, BLARP . . . " You wake up with a start to the loud ringing of your universal information appliance in your apartment. During the night, the overseas sales office scheduled an emergency meeting first thing in the morning. They checked your schedule and found that you had an open slot. (You gave them authorization to do this.) By scheduling the early slot in your day, your UIA subsequently changed your wake-up time appropriately (but within the permitted boundaries). Glancing at the time, you

Figure 1 An example of the UIA vision for personalized interfaces



pick up your UIA, turn off its alarm, and place the UIA in your pocket. While grabbing a quick breakfast in your kitchen, you remember that you want to set the VCR to record a program in the evening, so you take a quick jaunt into the living room. As you retrieve your UIA from your pocket, you realize that the "top level" kitchen applications have been replaced with living room and entertainment applications. As you select VCR, the UIA quickly retrieves your saved preferences. You add the program that interests you to the list of programs to record, and then submit your new settings. As you leave the apartment building, the household applications disappear from the set of active applications on the UIA, and new applications appear. You click on the transportation application and find that your bus will be late (a flat tire has been reported). You open the alternatives menu and find that you can still catch a taxi to the subway stop. Selecting "taxi" in the menu, you get instant confirmation that the taxi will arrive in two minutes at your location, determined by a quick location fix courtesy of the built-in Global Positioning System (GPS).

Implementation challenges

The challenges to achieving this vision are in three areas: (1) the user device with virtual machinery providing the UIA functionality, (2) the communication software infrastructure, and (3) the wireless communication link between the UIA and the infrastructure. Our initial implementation addresses these areas. For future implementations, we are focused on the complex and multifaceted problem of integrating the UIA into the mobile user's daily life. Our aim is to enable end-to-end flows of information, from the UIA to an arbitrary endpoint of communication, whether that endpoint is a peer UIA, a legacy data service, another application, or an electronic device. We also intend the UIA to be cross-platform; to this end, we are researching a range of wearable user devices to which to "port" the UIA capability. We discuss the core requirements of the user device and the infrastructure challenges in the two subsections that follow. We introduce the issues framing our current research in end-to-end data flow and alternative user devices in the third subsection.

Device requirements. The first requirement for the UIA vision is the user's portal, the device itself. Should the device have one common design, or should the UIA intelligence be integrated in all consumer items, from pendants and watches to wearable computers and PDAs? We assume that users will want to use a range of devices, and with decreasing hardware costs, users will eventually use computers as disposable commodities, assembling their machines from the most readily accessible components in a given situation. The safe assumption is that one device will not fit everyone's needs. As long as users expect flexibility among many devices, the best approach is to define the common elements that are needed in a UIA device—in other words, a reference platform. Then, regardless of the size and capabilities of a particular platform, any device that implements the required set of UIA functions will be a valid UIA. The required functions are as follows:

1. An output mechanism, most likely some sort of display—The display may be visual, tactile, or audio.[5] Ideally, a fully loaded UIA will have multiple output channels (visual, audio, and tactile) for more convenient (and natural) interaction.
2. An input mechanism—The input may be a touch screen (or something related to tactile input), a speech recognition unit, or even a visual sensor[6] (motion detector/gesture interpreter). Again, it is advantageous to have multiple input channels available.
3. Local data storage—There must be sufficient storage for the UIA engine, some number of cached interfaces, and some amount of data. It is a joint requirement that the UIA engine and the interface description language must be sufficiently compact, and the device must have sufficient storage. In our current implementation, the memory requirement of the UIA is 80 kilobytes. However, since even small devices, such as the Rex** Pro PDA,[7] have 500 kilobytes of memory, most platforms will have no trouble providing sufficient storage.
4. Network communication—The UIA must be able to externalize its actions in a generalizable way. The UIA can control a remote device, or be a client to a network service to the extent that the UIA can trigger activity in these remote entities, or be triggered by them. In order to "externalize" its activity, the UIA requires network communication that will relay data and event messages between the UIA and the devices and services with which it interacts. Additionally, the data types that the UIA exchanges may change with each new application scenario, so the network communication should be flexible to dynamic schema.

Another challenge is one of perceived function. Although a UIA will likely be turned off 99 percent of the time (actually a requirement from a power management standpoint), the UIA must appear up-to-date and ready with negligible startup time. When the UIA is powered on, it may be intermittently disconnected from the network. To remain always up-to-date, the UIA must periodically, and asynchronously, either contact a server for updates or have an alert capability by which the server can asynchronously establish a channel for pushing new information.

Communication infrastructure. In addition to the challenges of building the physical UIA device, there are challenges of building an infrastructure with which the UIA can communicate. The infrastructure includes communication middleware that connects the UIA to the various information, interface, and application servers. In addition, the communication middleware must possess a discovery capability to automatically connect the UIA to new domains when the user enters a new context (logical or physical), and numerous capabilities for the UIA to receive information relevant to the user's current context. The infrastructure must support query capability for the

UIA to look up information buffered by the infrastructure, a publish or subscribe capability for the UIA to receive information pushed from the infrastructure asynchronously, and an alert capability for the infrastructure to wake up the UIA when establishing a communication channel.

For user mobility, it is important that the UIA not be tethered; thus, the UIA must have a wireless link to the network. The wireless link can be infrared (IR) light transmission or radio frequency (RF) transmission. In the RF category, there are several options,[8] from the slower (6.4 KB/s [kilobytes per second] to 28.8 KB/s) 900 MHz (megahertz) paging and cellular voice systems, to the faster (nearly one MB/s) 2.4 GHz local area wireless networks.[9] The greater the variety of wireless modes by which the UIA can communicate with the network infrastructure, the better. For example, short-range IR, short-range RF, and long-range RF are all useful possibilities. In evaluating a wireless link technology for the UIA, it is important to consider the nature of the network communication of the UIA. The communication will consist of downloading new application interfaces and sending events and data packets (only a few bytes) during an application run time, in both wide-area and local-area scenarios. In the local area, the user may sometimes expect a connection-oriented session for continuous interaction with services and for media streaming, as well as short startup times. Thus, the short-range wireless link should provide adequate throughput to support a high message frequency and negligible application download times. In the wide area, the user's network computing will be more intermittent, so some message latency is tolerable, and application download times of a few seconds are probably adequate.

Seamless integration. The final, and possibly the largest, challenge is integrating the UIA seamlessly into the lives of the public. Rather than teach the public how to use the UIA, we aim to build a device that is a person's ready assistant, directing the electronic world on the user's behalf. In many cases, the UIA should appear to the user as simply a universal translator allowing the user to interact with intelligent devices. "TV, turn on," "VCR, record all showings of 'X-Files' this week," or "Radio, please remove any radio stations from the predefined list that feature overly conservative political hosts," are all examples of commands that a user could potentially generate. Though not part of our initial prototype, we believe that, eventually, the speech interface will become the predominant input mechanism. User devices will have other novel input and output modes such as tactile I/O, relative motion sensors, and projected displays. The UIA vision relies on users having the freedom to select from a range of user devices and to perhaps use different devices from one situation to another, while being guaranteed the same UIA functionality. Heterogeneity implies virtual machinery that will interpret platform-independent UIA interface descriptions into specific implementations. (Although our initial prototype of the UIA includes virtual machinery for only one PDA platform, the IBM WorkPad*, we are researching comparable virtual machinery for other user devices, such as badge and laptop computers.)

In some situations, the UIA will act as the user's extended self, gathering and presenting just the right information to augment the user in his or her current activity. A classic example is a weather beacon (suggested in a recent article in the popular press[10]): When the weather brings rain, the UIA suggests shops in the user's current proximity that sell umbrellas. The user perceives that the UIA is doing a great deal of helpful work on his or her behalf, as if the user were the one assessing the situation and gathering the needed information. In actuality, the "work" is the outcome of the UIA programming model. For example, when weather events are posted to the communication middleware, the middleware invokes a service for gathering information on retail in the user's location (using the user's real-time location, also posted to the middleware). When the information service returns with suggested stores, the middleware alerts the user's UIA and "pushes" the suggestions over the wireless link to the user's UIA interface. The end-to-end data flow between the application on the UIA and the data server, in response to the user's context, requires the capability to exchange messages containing data and state information through the infrastructure. However, this alone is not enough. The data flow requires intelligence to route the messages through the middleware asynchronously and through intermediaries that transcode between incompatible data formats, event models, and protocols at any point in the path.

Our first implementation of a universal information appliance

To implement our first UIA, it is necessary to develop a design and architecture for the implementation.

Summary of requirements. The UIA product that we envision in the mainstream (as suggested earlier) is

not quite the same device that we are implementing now, but it is similar in its basic function. Summarizing from the previous section, a universal information appliance must be able to receive device and program interfaces dynamically, render them, and react in the appropriate manner. (For example, send network message to invoke printer, send IR signal to TV, compute a value, or store a value in the database.) The UIA must also be able to store program and user data in a local database. Additionally, the UIA must be able to communicate with network middleware that passes requests through to appropriate applications, and buffers data intended for the UIA. The communication mechanism should allow the user to roam transparently between computing contexts. Thus, the UIA must support a wireless link, and the wireless infrastructure should support seamless roaming between short- and long-range connections.

Implementation architecture. In this subsection, we describe the UIA device and communication infrastructure that we have designed in phase one of our project in view of the stated requirements. For the UIA device, we have developed a platform-independent application and interface language and a local storage system. These are implemented on a standard PDA (the IBM WorkPad, also made as the 3Com PalmPilot**). For the wireless connection, we use an existing wireless messaging infrastructure to deliver information over long distances, and IR and high-frequency RF for short-range communication. For the communication middleware we use TSpaces* (formerly written as "T Spaces"), middleware developed at IBM, that "glues" system components together by combining data management, computation, and communication. Although this initial implementation uses a specific hardware platform, the solution is intended to be hardware-independent.

UIA device architecture: Building the UIA using the IBM WorkPad. From a software perspective, the UIA requires three major components:

1. A platform-independent application and interface language that can be efficiently retrieved dynamically over a wireless connection
2. A local on-board database that stores application data
3. A network interface to communicate with the electronic universe and a local cache mechanism that can be used to store application data intended for the network when the device is disconnected

MoDAL language. Since binary representations of applications vary with differing hardware platforms, the design of the UIA calls for a platform-independent representation of applications and interfaces. Interfaces, as highly structured entities, are well-suited for encoding in a description language. We have chosen the current standard language in industry for structured document description,[11] the eXtensible Markup Language (XML),[12] to represent UIA applications and interfaces. Since the application descriptions are relatively high level, this approach gains a level of abstraction that makes UIA applications and interfaces sufficiently compact to be retrieved dynamically over the network without bloating limited network resources. To illustrate this point, consider the simple "Credit Me" application shown in Figure 2. This application, written in 20 lines of MoDAL, requires over a thousand lines in a traditional imperative approach, such as the C programming language. Our interface and application language, defined in XML, which we have named the *Mobile Document Application Language* (MoDAL), allows the application designer to create highly complex applications.

MoDAL offers extensions to support dynamic user interface creation, network service access, and local database access, as well as flow control structures and local variable assignment. The result is that MoDAL interface content and style are dynamically configurable as a function of data received from the network middleware, and the user's interaction with interface resources (widgets) can asynchronously communicate data and application state to the middleware. Additionally, the MoDAL language is designed to allow mobile clients to easily and efficiently retrieve updates to applications without having to retrieve a full copy of the updated application. Thus MoDAL applications—very small, powerful document-based applications—serve as dynamic interfaces to the electronic universe, exchanging data and events between the UIA and any peer, and being updated with new interfaces as the mobile user moves contexts.

Local database. To extend the notion of the UIA beyond a glorified remote control, the UIA must be equipped with a local database. This local database allows the user to configure and personalize interfaces, along with defining certain behaviors, which are uploaded to the intelligent middleware (e.g., "always turn on the lights when I enter the house"). Additionally, personal heuristics such as voiceprint, encrypted passwords, etc., can be stored in the UIA for

security and verification purposes. In the soda machine example of the "Credit Me" application in Figure 2, we can transmit information identifying ourselves to the soda machine, which can then charge us for the soda, retrieving our billing information from a network database. It is a very simple application written in MoDAL as an application interface to a soda machine. This application consists of a single "form" or user interface that displays a simple button with the text "Credit Me." Pushing the "Credit Me" button sends a message to the soda machine that causes the machine to give the user a free soda. How does the system know whom to charge for the soda? The answer lies in stored user profiles—both on the UIA and in the network. The MoDAL code for this application is shown in Appendix A.

Figure 2 An example of the "Credit Me" application

Equipped with a MoDAL engine, a local database, and a wireless network connection, the UIA paradigm is very appealing. Moving from location to location, the UIA retrieves a list of available interfaces for the current location. Data and control messages are passed wirelessly to the various devices, enabling the user to interact with his or her environment.

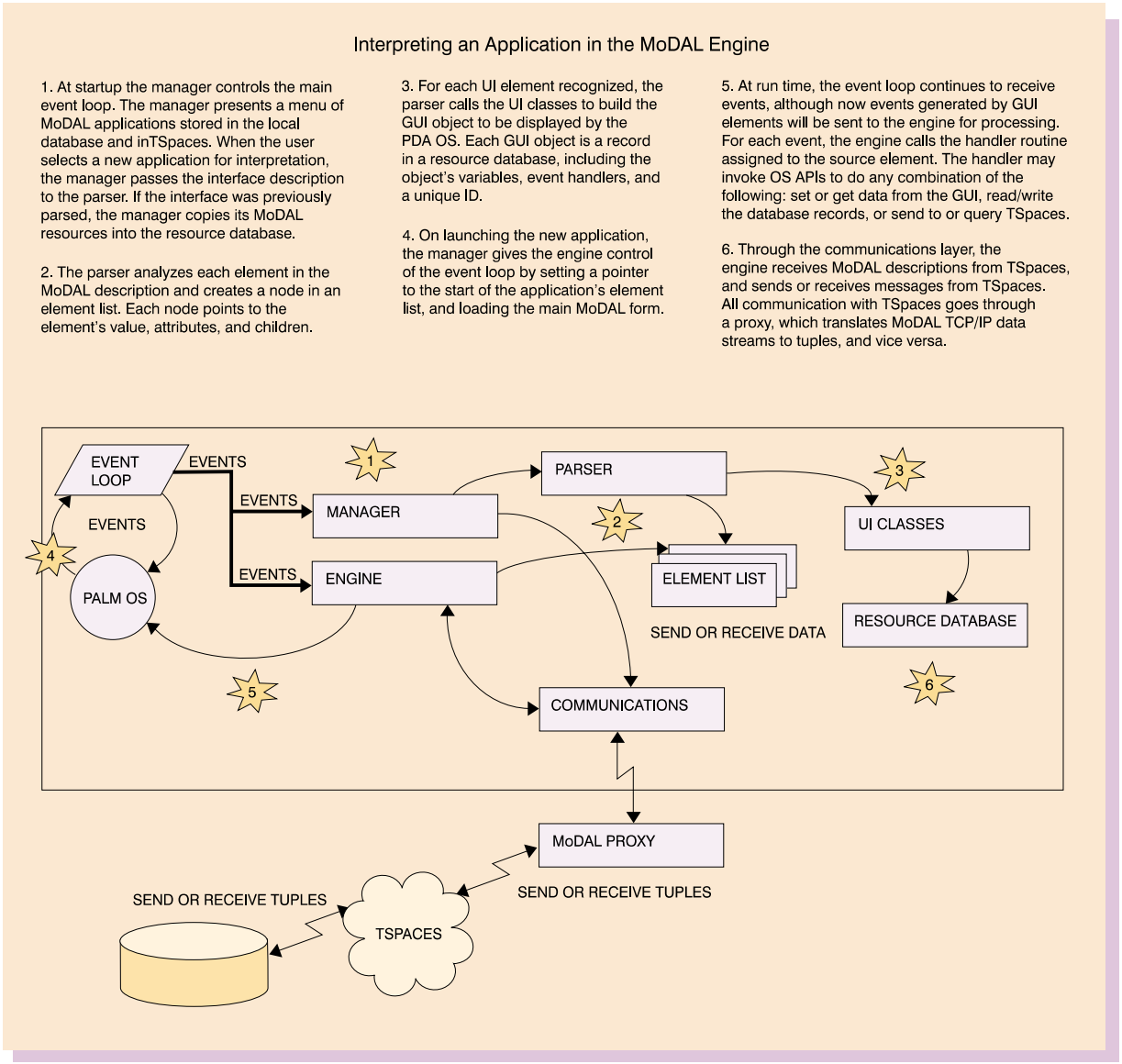
The UIA engine architecture. As shown in Figure 3, the UIA engine consists of the interface manager, the MoDAL interpreter, an XML parser, and associated databases. The interface manager is responsible for retrieving new MoDAL interface descriptions and presenting the user with a list of available interfaces (label 1). When a new interface is discovered (downloaded to the UIA), the MoDAL description is passed to an XML parser that generates the elements (local variables, event handlers, etc., label 2) and user interface components necessary for the MoDAL application (label 3). When a MoDAL application is "launched" from the interface manager (label 4), the MoDAL interpreter performs a lookup on the referenced application and proceeds to load and create the corresponding user interface. Additionally, the UIA engine creates the corresponding event handlers from the information stored in the interface elements database. While running an application, the MoDAL interpreter may make calls to the PDA operating system (OS) database application programming interface (API) or networking API (label 5), or both.

The two data resources (the interface elements and the resource database) created by the parser serve as the core of the MoDAL engine. The resource database is a compiled version of the user interface (UI) elements included in the MoDAL description (label 6). Available UI elements include forms, buttons, text fields, menus, labels, lists, tables, and help strings. The parser builds the compiled resource by dynamically creating the UI element data structures,[13] and then copying the data structure into a resource that is accessible by the PDA OS Form API.

The second data resource is a list of interface elements. This resource represents the kernel of the MoDAL interface; it stores names, values, and relationships between elements. The members of this resource, *resourceID*, *token*, *valueType*, *value*, *next*, and *attr*, are described in more detail in Appendix B.

Defining the MoDAL language. MoDAL allows the IBM WorkPad to perform four main actions: present a graphical user interface, perform local computation, read or write local database data, and send or receive messages to the communication component (TSpaces). The user interface is generated dynam-

Figure 3 The UIA engine with a description of run-time functions



ically by the MoDAL interpreter, instead of being created externally to the device by a resource compiler. A complete description of the MoDAL language can be found in a separate document.[14]

The current implementation focuses directly on the WorkPad. Consequently, the MoDAL description contains WorkPad-specific values (i.e., a 160 × 160 pixel-display, and the Palm OS** widget set). However, we plan to change this focus in upcoming versions, that is, we aim to make the MoDAL language platform-neutral. To this end, we are considering the incorporation of Extensible Style Sheet Language (XSL) support into the MoDAL specification. The XSL will define how a particular MoDAL description should be mapped to a particular device, and thus, allow the MoDAL language to remain device-independent.

Figure 4 is a simple "Hello World" application written in MoDAL. The running application displays a single button labeled "Say Hello"; when the user clicks on the button, the WorkPad responds with "Hello World." The example source starts by defining an application named "Hello." Inside the application, one form—"MainForm"—uses the entire 160 × 160 pixel screen of the WorkPad. Within the form is one textfield, which is used by the application to display its message. The definition of the textfield includes the name of the textfield, its dimensions, and placement. Additionally, the textfield definition sets the character length of the textfield and defines the number of lines of text to be displayed—one or more. The next element is a button. The button definition has a name, a text label to be displayed in the button, dimensions, and position coordinates (that set its position to the top left corner). Additionally, the button has an associated *action*. Actions are event-handlers that respond to events originating from the parent UI element. The handler in this example issues a SET command, setting the value of the textfield to the appropriate string when the button generates an event (is clicked).

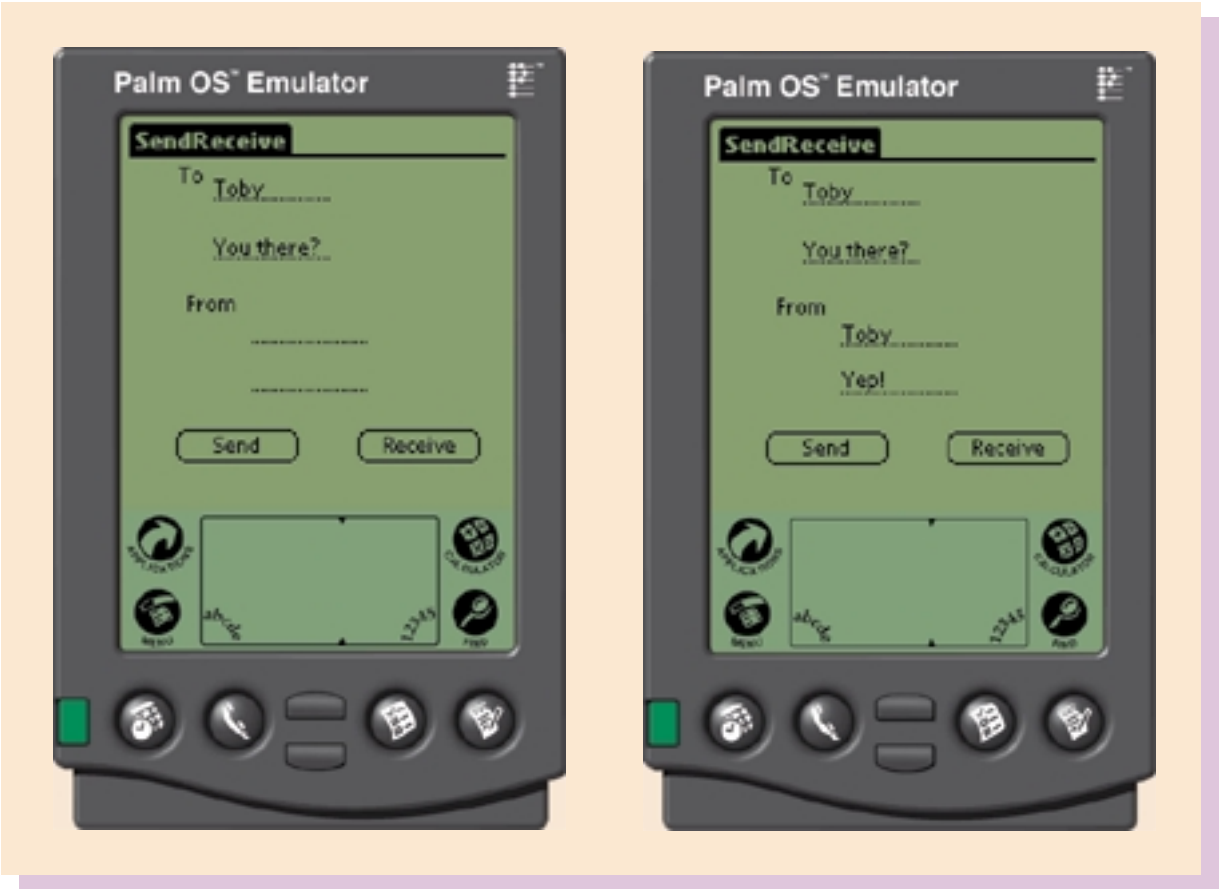
As a slightly more complex example, Figure 5 shows a MoDAL application that interacts with TSpaces, the underlying communication layer. This example will send and receive text messages to and from TSpaces. Each message is composed of a username and an associated text message. Upon execution, the MoDAL interpreter displays a form with four textfields, a label, and two buttons. The first pair of textfield elements are for the user to enter a user name and a text message to send; the second pair are for displaying an incoming message's user name and text, respectively. When the user clicks on the "Bsend" button, the action associated with the button generates a tuple composed of the strings in the "TFuser" textfield and the "TFmsgToSend" textfield. Clicking on the button "BRcv" will generate a TQUERY that queries TSpaces for a tuple with two fields, each containing data of type string (STRING*STRING), and sets the text fields "TFfrom" and "TFmsgRcvd" with the data returned. The user sees the sender's name displayed in a textfield labeled "From" and the chat message displayed in a textfield below.

TSpaces: The delivery mechanism. We use TSpaces[15] as the communication middleware between the UIA and the network applications that house the majority of the user's data. TSpaces is particularly suited to this task. TSpaces is a network communication buffer with database capabilities that enables communication between applications and devices in a network of heterogeneous computers and operating systems. That is, TSpaces allows any program or device to locate and communicate with any other program or device, regardless of hardware, computing platform, or location (Figure 6). TSpaces provides group communication services, database services, URL-based file transfer services, and event-notification services.

Figure 4 A simple MoDAL application

TSpaces background. Structurally, TSpaces is a lightweight database system[16] coupled with a tuplespace[17] communication system, written in the Java** language. TSpaces, like all tuplespace systems, uses a shared whiteboard model (all clients can see the same global message board, as opposed to multiple point-to-point communication). Clients can see tuples, posted by others, by issuing queries containing cer-

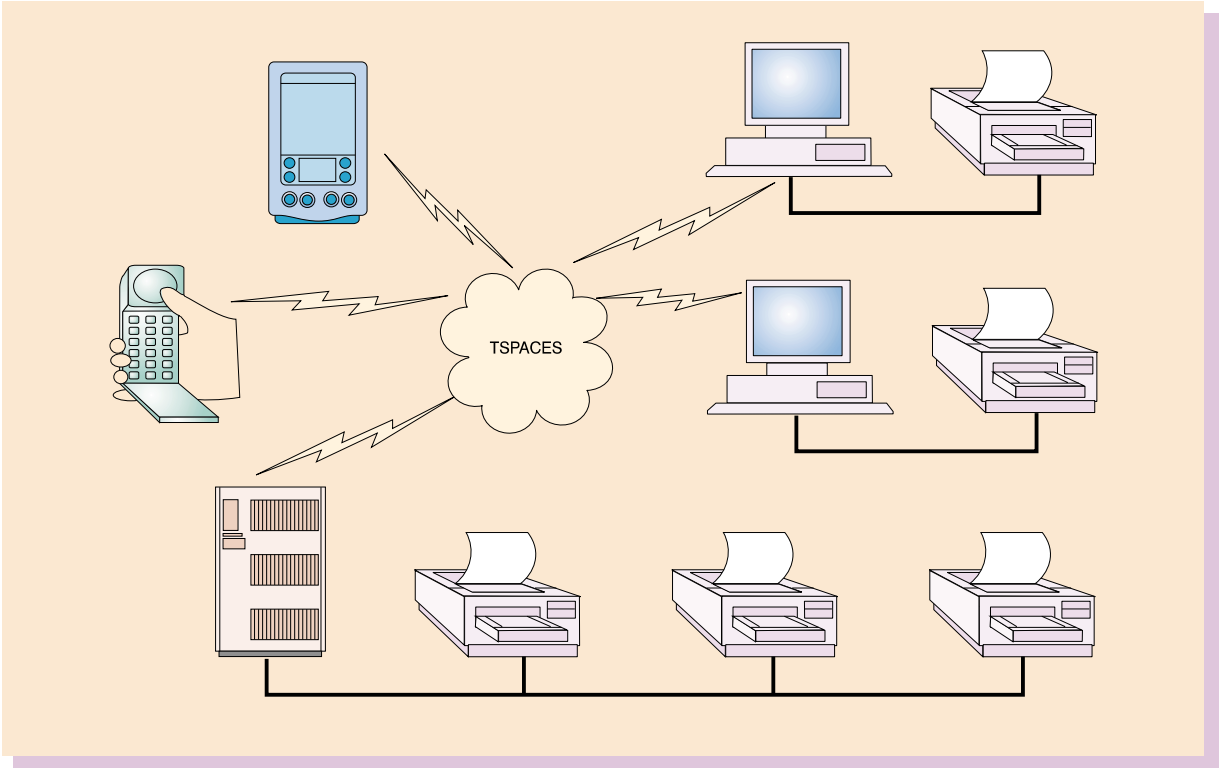
Figure 5 A simple chat client application in MoDAL with Tspaces interaction



tain filters; for example, read "all of Harry's posts" or consume "all posts related to printing." Because TSpaces is implemented in Java and uses protocols with standard TCP/IP, TSpaces is an ideal middleware component for making network-oriented services available to any client, regardless of the computing platform. For example, a network service such as printing, e-mail, network fax service, remote device control, or program translation can be implemented using TSpaces in the following way. A client in need of a service, such as a PDA e-mail client, simply sends a message in the form of a tuple[18] to the TSpaces server. The tuple specifies the service needed (e.g., e-mail), the data to be processed (e.g., the header and body of an e-mail note), and any essential application state (e.g., the timestamp of the message). A service provider application (e.g., an e-mail gateway) registers an interest with the TSpaces server for all tuples, mentioning that particular service class (e.g., e-mail). When a tuple mentioning that service class appears, the TSpaces server notifies the service provider, whereupon the service provider removes the tuple and processes it (e.g., packages the e-mail and routes it to the designated recipient).

The rewards for this simple model are many. First, this model uses the existing computing infrastructure—there is no need to change hardware, and only minor, one-time software modifications in the form of client interfaces are required. Second, the model standardizes the interfaces to all services for all platforms—it creates a uniform layer on top of the various heterogeneous platforms. Third, new services (or new clients, for that matter) can be added dynamically without reconfiguration (or recompilation) of the servers or clients. Finally, and perhaps most im-

Figure 6 TSpaces, the communication middleware for the UIA



portantly, TSpaces, being a database system, can maintain and publish the active set of resources or services so that clients may discover them *automatically*. For example, by making a standard directory service, a client of TSpaces, such as Lightweight Directory Access Protocol (LDAP),[19] Service Location Protocol (SLP),[20] or Domain Name Service (DNS), and by storing the contents of the directory in TSpaces, an application can ask TSpaces for a resource reference using the standardized (well-known) protocol. We plan to have future versions of TSpaces include interfaces for well-known directory service protocols, as well as the capability for the user to customize an interface.

Though very much like a relational database in function, TSpaces has a much simpler data model and query language. Thus, clients can easily store and retrieve data without having to first define any tables, normalize any data, or learn any complicated query languages. However, though simple and easy to use, TSpaces does have a rich feature set. TSpaces has update and query operations, administrative control, transactions, application isolation, and direct support for UIA-like devices. Details on the TSpaces system can be found in a recent *IBM Systems Journal* paper.[15]

TSpaces as messenger, database, and file system. For the UIA, TSpaces performs several services. First, TSpaces is a messenger service, connecting the UIA to any service it might need, such as printing, faxing, e-mail, search services, Web proxy services, remote device control, remote (legacy) application invocation, or program translation. In addition, TSpaces manages the connection of the UIA to other UIAs (PDAs) when a direct peer-to-peer link is not possible, either because the peer is remote or has an incompatible underlying platform that requires an intermediary for communication. Second, TSpaces is a file system for the UIA—TSpaces is the base with which the UIA synchronizes its data for backup and from which the UIA downloads and launches interfaces for mobile applications. In our implementa-

tion, TSpaces augments the native communication and application storage models of the UIA, requiring only a lightweight proxy to transcode messages from the UIA implementation language to TSpaces. Third, TSpaces is a resource directory service and database for the UIA, serving names and resources for people and services. Finally, TSpaces acts as an intermediate database system, caching data from network and mainframe database systems and legacy applications.

The wireless infrastructure and two-way wireless communication link are essential in the overall UIA architecture.

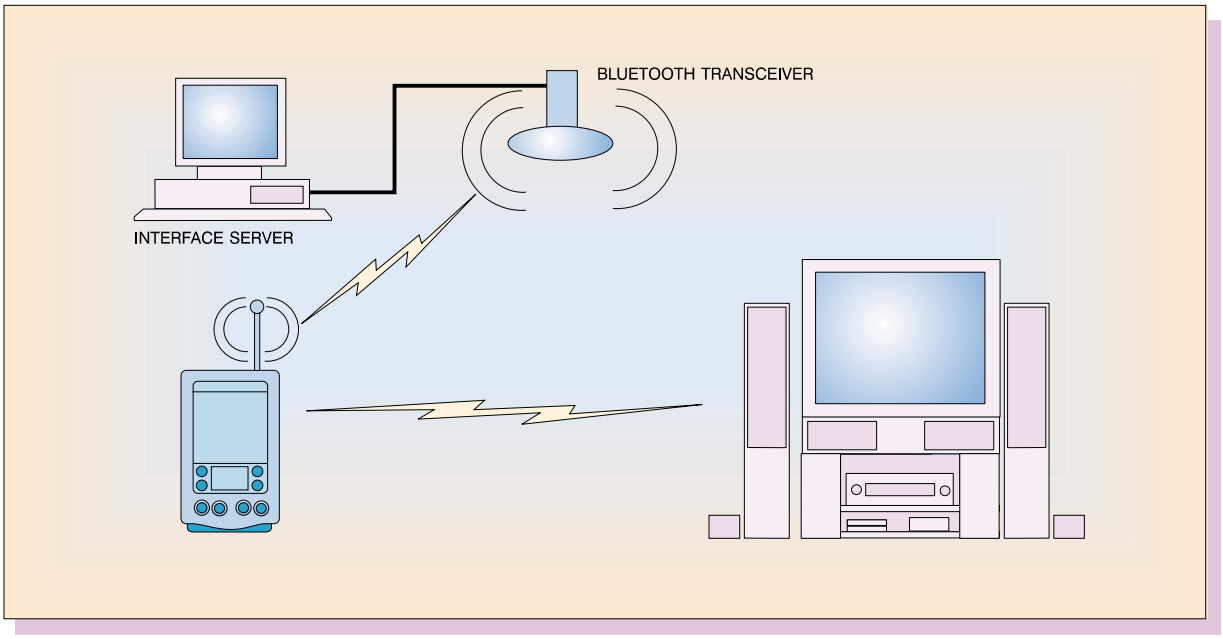
The wireless connection. The UIA vision for context-aware interfaces depends on mobility for the UIA and the ability for the UIA to discover devices and services in its physical proximity. Thus, the wireless infrastructure and the two-way wireless communication link for the UIA are essential in the overall UIA architecture. The wireless link for the UIA has two modes: high-speed, short-range mode and low-speed, long-range mode. An important use of the high-speed short-range mode is device discovery. For example, using a discovery protocol, such as Service Location Protocol,[20] Salutation,[21] or Jini**,[22] when a user with a UIA enters a "smart" room, the UIA picks up a broadcast signal declaring services are available. The UIA responds with its ID and queries the room for applications that it might download in order to interact with the appliances or media services of the room.

When operating in "high-speed mode," the UIA is able to function as a network computer. That is, it has real-time access to the available network services (printing, e-mail, fax services, remote device control, program translation, etc.). The network services download their interfaces over the wireless connection, whereupon the UIA can use them to execute the various services. Two possible wireless technologies for the high-speed mode are an infrared link, such as that used on the IBM WorkPad model 8602-20x[5], or a radio frequency link transmitting at frequencies in the upper RF band (1–2 GHz), such as the proposed Bluetooth technology[9] (Figure 7). However, both technologies have limited range—on the order of 10 meters. Infrared (IR) has a range of 1–10 meters and Bluetooth (BT) has a range of 10–100 meters.[9] Because of this distance restriction, "corridors" for high-speed access are essential in well-populated areas where service offerings are dense, such as places of work or university campuses.

At times, a user will travel outside the range of all high-speed corridors. Even so, the user can still connect to the network through a "low-speed mode," which exchanges data throughput for range. The low-speed mode is implemented using TCP/IP over, for example, a two-way messaging system, such as the reFLEX protocol from Motorola,[23] for data rates around 14.4 KB/s (though current data rates for reFLEX are slower—9.6 KB/s for the uplink and 6.4 KB/s for the downlink). Alternate wireless data technologies include data over voice cellular solutions, such as Cellular Digital Packet Data (CDPD, wireless IP), Code Division Multiple Access (CDMA), and Time Division Multiple Access (TDMA) (both digital cellular), and Global System for Mobile communications (GSM, the standard for wireless voice and data in Europe and Japan), as well as radio modems (e.g., Ricochet) (Figure 8). However, we are focusing on the two-way pager infrastructure as a cost-effective near-term solution. The reFLEX pager infrastructure can potentially support almost complete coverage for connectivity, as most areas of the United States are already covered by a combination of pager networks.[24]

The intentions of having the high-speed and the low-speed wireless modes for the UIA are based on what the user is likely to require in short-range and long-range situations, respectively. To make a comparison, whereas the high-speed link is the "interface connection" for the UIA acting as a network computer, the low-speed link is the "data connection." The low-speed link is the channel by which the user receives information updates to programs, such as e-mail, messages, calendar updates, and other asynchronous, application-specific updates. It is possible to download applications over the low-speed link. For example, a simple MoDAL application (a few frames and 10–15 actions) is only a few lines of code, on the order of 1–5 KB. Over a high-speed connection, the application download is practically instantaneous; over a low-speed connection, the download time is only a few seconds. A complex MoDAL application (50 frames, 200–300 actions), for example,

Figure 7 The short-range wireless connection—PDA interacts with entertainment center, having received interfaces from local interface server



could be on the order of 100 KB. Over the high-speed link, the application download time is still negligible. Over the low-speed link, assuming an average data rate of 20 KB/s, the download time is 40 seconds—some latency but still tolerable. However, the absence of a specific context (such as the proximity of a device) and the latency make the slower link impractical for applications with strict real-time requirements, such as interfaces for fine-grained remote control.

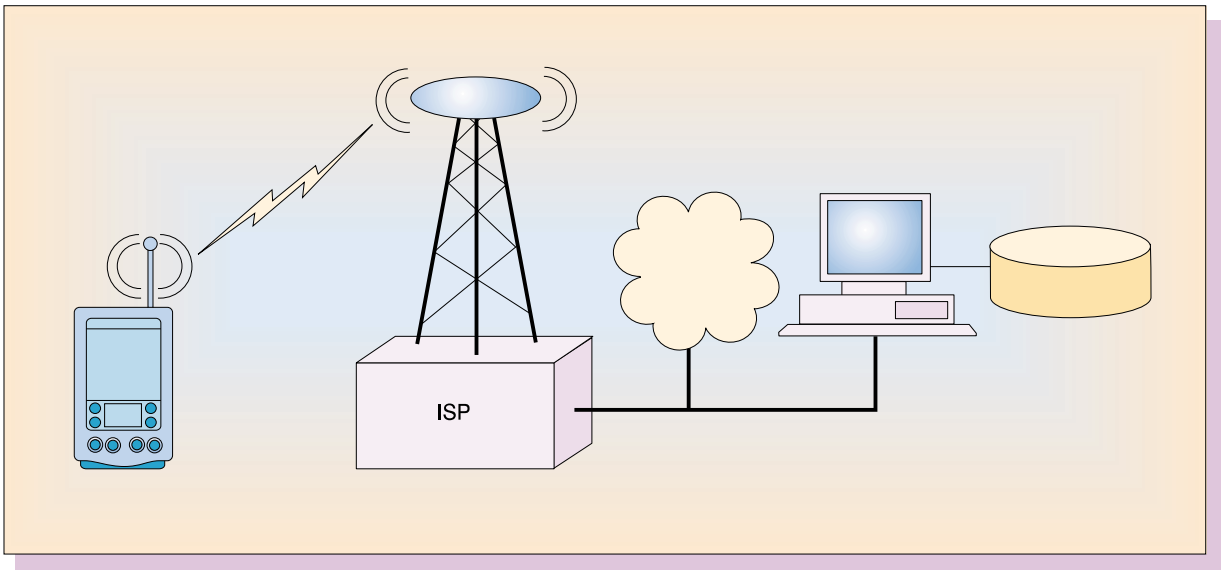
A popular application—The active calendar

One of the most popular applications for the PDA, and consequently, a useful application for the UIA, is the calendar. The typical PDA calendar provides the user daily reminders of planned activities; however, the information provided is static, consisting of facts the user has entered and of which the user is already aware. The active calendar application[25] aims to change this by combining vast available information with specific personal needs in a new convenient way. In addition to storing the information the user inputs, active calendar automatically uses that information to predict what additional information may be needed. The system is able to search and retrieve information from a multitude of sources (e.g., a local disk, an organization's intranet, or the Web), organize the information, and bring it to the user. A user can customize the information to be retrieved and the actions to be taken in response.

For example, assume a user enters "meet with John D at XYZ-Soft about Project-X." When the entry is stored, the active calendar service collects information about XYZ-Soft (an up-to-date stock quote, company news, etc.), information about John D (e.g., telephone number, e-mail address), and about Project-X (e.g., project home page and general Web search results). Then active calendar links this information to the calendar entry.

We envision future PDA calendar applications being extended with the "active" features in this example, relying on the UIA and its communications infrastructure as enabling technologies. An active calendar service (see the next subsection) simply registers itself as a service provider application in a TSpaces database system, and exposes its interface by registering its interface methods with a directory service integrated with TSpaces. A UIA-enabled PDA calendar application can now be extended to take advantage

Figure 8 The long-range wireless connection



of this new service and automatically augment the calendar entries with helpful "just-in-time" information (see the next section). A final but nontrivial point: because active calendars "predict" and automatically collect information that the user might need in the future, waiting to download the information when a user connects, active calendars are ideally suited to work in disconnected mode.

Active calendar implementation. The current implementation of an active calendar service at the IBM Almaden Research Center is operational. The system is capable of automatically searching and collecting information for the Lotus Notes** calendar system, which is being used by most of IBM's employees worldwide. To maximize portability, the active calendar code (approximately 22000 lines) is written in Java. The active calendar service currently runs on a single server and automatically accesses and augments calendar entries for one dozen test users. The current implementation focuses on automatic retrieval of information typically needed by IBM Almaden employees.

We have compiled a list of more than 30 events commonly used by employees at Almaden. These include various kinds of meetings (e.g., department meetings, round-table meetings, invention meetings, etc.), events that involve travel, such as attending a conference, and private events, such as birthdays and vacations. For each event, we have defined a set of default actions that satisfy the information needs of typical Almaden users. As an example, an Almaden project meeting has the following information: people attending the meeting, the subject of the meeting, and the location (e.g., conference room) where the meeting takes place. The active calendar automatically collects the following information:

- Contact information, such as telephone number, e-mail address, office location, and job responsibilities, for each person attending the meeting
- Project home pages and a brief project description, obtained by querying a project database for the subject(s) of the meeting
- All the reservations of the conference room where the meeting takes place on the day of the event, retrieved from a Lotus Notes database that stores conference room reservations at the Almaden Research Center

Another example is a business trip, where the user provides information on the destination of the trip (city, state, and possibly an address) and, if necessary, a preferred airport. The active calendar returns the following information:

- A timetable listing all flights from the user's home city to the destination on the particular dates of the event
- Driving directions from the airport to the destination
- Hotels and restaurants in the neighborhood of the destination
- Any events in the destination city occurring during the visit
- A weather forecast for the destination, updated daily and starting five days before the actual event

We are at the initial stage of deploying the active calendar at the Almaden Research Center. Many challenging problems remain to be solved, among which are usability issues and how to improve the quality of the information collected. However, we are convinced that the calendar is truly the right channel for pushing information to people who typically are too busy to track down the information themselves. Currently we are improving the user interface on the Lotus Notes and Microsoft Outlook** calendar systems, which ultimately will provide us with an application that is easier to use and customize and that is able to collect more accurate information.

Integrating the UIA and active calendar—Extending the UIA for "push"

Upcoming versions of active calendar will be integrated with the UIA, so that PDA calendar applications written in MoDAL, or augmented by MoDAL helpers, can interact with the active calendar service through TSpaces. Integrating the active calendar service and UIA to deliver tailored information represents a central theme in our present research: extending the UIA for information "push."

We consider the first version of the UIA—a MoDAL engine executing on an IBM WorkPad—as a glimpse into the next generation of computing. The popular media is abuzz with projections of a fundamentally new computing experience characterized by a pervasive "push" of information to the chameleon-like user interface, ever adjusting itself to the push stream.[10] In the envisioned scenarios, the interface style and content are always tailored to the user's context, for example, the user may step into a new place and download just the right interface to reach out to the available devices and services. Such scenarios require MoDAL capability to build soft application interfaces on the fly. We realize, though, that soft interfaces are only the first step. In order to push the right information to the user when the user needs it ("just in time"), the present UIA architecture must be extended such that MoDAL applications openly interoperate with other applications and services in the network.

The present implementation of MoDAL employs a "pull" model in which the MoDAL client can query TSpaces for events sent to TSpaces by other active clients. In order to realize the UIA vision, the MoDAL architecture must be extended for "push" of events. This extension requires opening the MoDAL and TSpaces event model to events detected by, or internal to, other active components in the network. In this section, we illustrate this and discuss our current research in using events as the basis for composing UIA applications. First, in the following subsection we explain why "pushing" information and interface style to the UIA based on the user's context relies on the capability to exchange events among the UIA and other active networked components through the middleware (TSpaces). We show this with the example of a remote control application for the UIA. Then, in the subsequent subsection we explain how we are building on the event exchange model to create an architecture for UIA applications in which MoDAL clients universally interoperate with peer applications and devices through TSpaces. Finally, in the last subsection we outline the design for the UIA active calendar application that we are currently building, using the event exchange approach.

Achieving context-aware push with event exchange. In traditional, sequentially programmed applications, software components can only "listen" for and respond to events within the specific event model of the application, that is, events that are of a compatible type and which are generated by well-known objects, internal to the application. "Pushing" context-dependent state and data to the UIA requires an open event model. UIA applications must be able to trigger off events generated externally by other active components. Similarly, in order for the UIA to interoperate with a remote component, such as a device or application service, the events generated in a UIA application must be externalized to trigger behaviors in the remote component. Assuming a dynamic user-context, the event types that the UIA application will encounter are unpredictable, so UIA applications must be designed to incorporate new event types at run time. Finally, to tailor UIA application content and style to the user's context, the programming model for UIA applications must allow the programmer to translate events inferring the user's

context into meaningful application behaviors (content and style updates). The range of event types this programming model must encompass is broad and extensible. In fact, any instance of activity, which can be digitized and communicated as a typed bit stream (message) is potentially an "event." For example, events may include the user's interactions with an application, or activity detected by the many sensors deployed in the physical world giving hints about the user's location, proximity to equipment or people, and current behavior. To meet these requirements, we are taking a new approach to building distributed applications for the UIA that relies on the middleware (TSpaces) to communicate events between the UIA and peers with which it interoperates.

We assume events are the glue for composing distributed applications.

To illustrate, consider a UIA application for controlling a factory floor machine from the UIA. First, upon entering the factory floor, an employee is presented with the opportunity to download the MoDAL application for the machine. This action requires the network middleware (a TSpaces server), on behalf of the UIA, to interact with a user location service for the factory area. Depending on the accuracy of the location required by the application, the location service may comprise any number of technologies: a wireless network gateway, an electronic badging system, or a system collecting signals from passive RF receivers ubiquitously deployed around the factory.[26] The TSpaces (TS) server has registered for events indicating the presence of this particular user. When the user enters the environment, the location service sends an event identifying this user to TSpaces. In turn, the TS server signals the user's UIA to launch a MoDAL application appropriate to the circumstance, for example, an application for downloading machine controls.

The application presents the user with the MoDAL interfaces for the local domain (the factory floor) stored in TSpaces. (Alternately, the interfaces may be stored by an interface server connected to TSpaces.) When the user selects the machine control application, his or her UIA connects to TSpaces and downloads the MoDAL description from TSpaces over the wireless link. Concurrently, the TS server registers the UI objects on the UIA with the corresponding objects in the software controls for the machine, and with databases or message services providing information about the machine. For example, the TS server registers the MoDAL UI elements for events from the corresponding software controls of the factory machine, and vice versa. Thus, the user can activate the machine from the UI controls on his or her UIA and receive feedback in the form of text in dialog boxes on the UIA, as if he or she were actually using the control panel of the machine. Similarly, the TS server registers a MoDAL dialog box for messages about the factory machine from a mail gateway or an inventory database. This allows messages pertaining to the machine to be pushed to the MoDAL dialog box.

In these examples, the TS server makes event registrations on behalf of clients, e.g., the MoDAL application and the other agents with which MoDAL will interoperate. When events matching these registrations occur, the TS server is notified by the event-notification mechanism built into TSpaces. It is then up to the TS server to communicate these events to the MoDAL client as if the events happened internally to the client. This approach bypasses the process of translating remote events into local events. Instead, the TS server simply invokes appropriate actions defined by the MoDAL client in a generic interface, passing parameters from the notified event. The TS server translates the notified event into the appropriate action by dynamically interpreting stored event rules. We describe an event-processing engine in TSpaces that performs this function in the next subsection; also see Munson.[27]

In the factory floor example, when an e-mail gateway receives a message pertaining to the factory machine, the gateway sends an event to TSpaces. Having registered for e-mail events pertaining to the machine, the TS server is notified. The TS server processes the event, selects the appropriate client (the user's MoDAL application), and assembles the appropriate action object. In turn, the TS server performs a lookup of the action interface for the client and invokes the corresponding action method. The implementation of the action displays the e-mail on the UIA by, for example, setting the value of a textfield object. The result is that the user sees the e-mail text on his or her UIA.

The same concepts apply for a MoDAL active calendar. The TSpaces middleware registers the MoDAL objects in a PDA calendar application for pertinent data located by the active calendar service. Registration takes place in real time so that the information pushed is always tailored to the user's most current situation. For example, by monitoring real-time location events such as those received from a GPS receiver, the active calendar service can push driving directions turn by turn as the user travels to a meeting.

Our current work focuses on how the TSpaces middleware can exchange events between a MoDAL application and distributed services and devices, even when the event models of the interacting components are not compatible. We intend TSpaces to be the pervasive event listener for any client, such that TSpaces is programmed to listen for events from external components and invoke the appropriate client actions when external events occur. Thus, events (in TSpaces) are the basis for composing distributed applications in which MoDAL clients interoperate with any other component.

An architecture for building UIA applications using event exchange in TSpaces. Our approach to achieve universal interoperability for UIA applications assumes that events are the glue for composing distributed applications. We use a broad definition of "event": An event is simply a unit of activity, either internal to or detected by an application component, that can be expressed as a parameterized message (tuple). UIA applications composed from distributed components interoperate by exchanging events through TSpaces. The event exchange between MoDAL clients and other components, such as databases, remote devices, and application servers, is programmed with a generalized grammar of event-triggers-action rules. "An event in entity A triggers an action in entity B." The basic expression in our rule grammar is:

```
Rule = TSClient.EventClass(parameters)
       → TSClient.ActionClass(parameters)
```

The TSpaces engine manages the event exchange according to these rules, at run time. Upon interpreting a new event rule, the TSpaces engine registers for the specified event on behalf of the TSpaces client. When the engine receives a notification that an event matching its registration has occurred, the engine searches its store for rule(s) matching the notified event. When it finds a match, the engine follows the rule and injects the specified actions in the appropriate receiving clients through generalized action interfaces.

In addition to the event-rule processing in the TSpaces engine, this model relies on four requirements: (1) a universal syntax for event and action classes, (2) action interfaces, opening clients for action invocation in response to external events, (3) an appropriate event model within TSpaces, including published interfaces for event registration and notification, and (4) transcoding proxies, to allow heterogeneous (non-Java) clients to notify TSpaces of events and to be invoked by TSpaces. These requirements are now described in more detail.

1. Universal event syntax—MoDAL, and the services and devices with which MoDAL applications interoperate, have internal representations of events. To mask incompatibilities, we are designing a universal object syntax for expressing events in XML. The extensibility of XML is convenient for designing hierarchical classifications of event types. For example, a general location event class representing location fixes for people or things, Location(entity, domain), can be subclassed for fine-grained location fixes, such as `GPS(entity, latitude, longitude, altitude)` and `Bluetooth(entity, domain, picocell)`. In addition, XML is becoming a popular metadata language for exchanging information between application domains, so an XML syntax for events increases the opportunity for interoperability.

2. Action interfaces—In order to "open" TSpaces clients for triggers from external events, active clients implement generic interfaces for action injection. For example, the actions in a MoDAL application to be triggered by external events are exposed through a generic object interface, published to a well-known directory service.

3. TSpaces event model—The event model within the TSpaces engine uses a publish-register-notify approach.[28] Each tuple space publishes an interface exposing the event classes contained. As the TSpaces engine interprets a new event rule, the engine performs a lookup of tuple spaces for events of that class and registers with those space(s). In registering with a space, the engine passes a tuple template specifying the event parameters (a variable or an exact value) to be matched. When an event matching the registration template is written to the space, the engine is called back (notified) and passed the matching

event. The matching of events to registration templates builds on the current TSpaces capability to match a tuple field on an exact value or any value of the field's type. We are adding registration interfaces for discovery of event classes by tuple space.

4. Transcoding proxies—A new generation of flexible proxies is required to interface TSpaces with the variety of potential MoDAL-like interpreters possible for mobile or wearable platforms. We are exploring IBM's Web browser intelligence (WBI)[29] technology for flexible proxy building. For example, future versions of the UIA may communicate with TSpaces over HyperText Transmission Protocol (HTTP), via a Web server, using a transcoding proxy such as WBI.

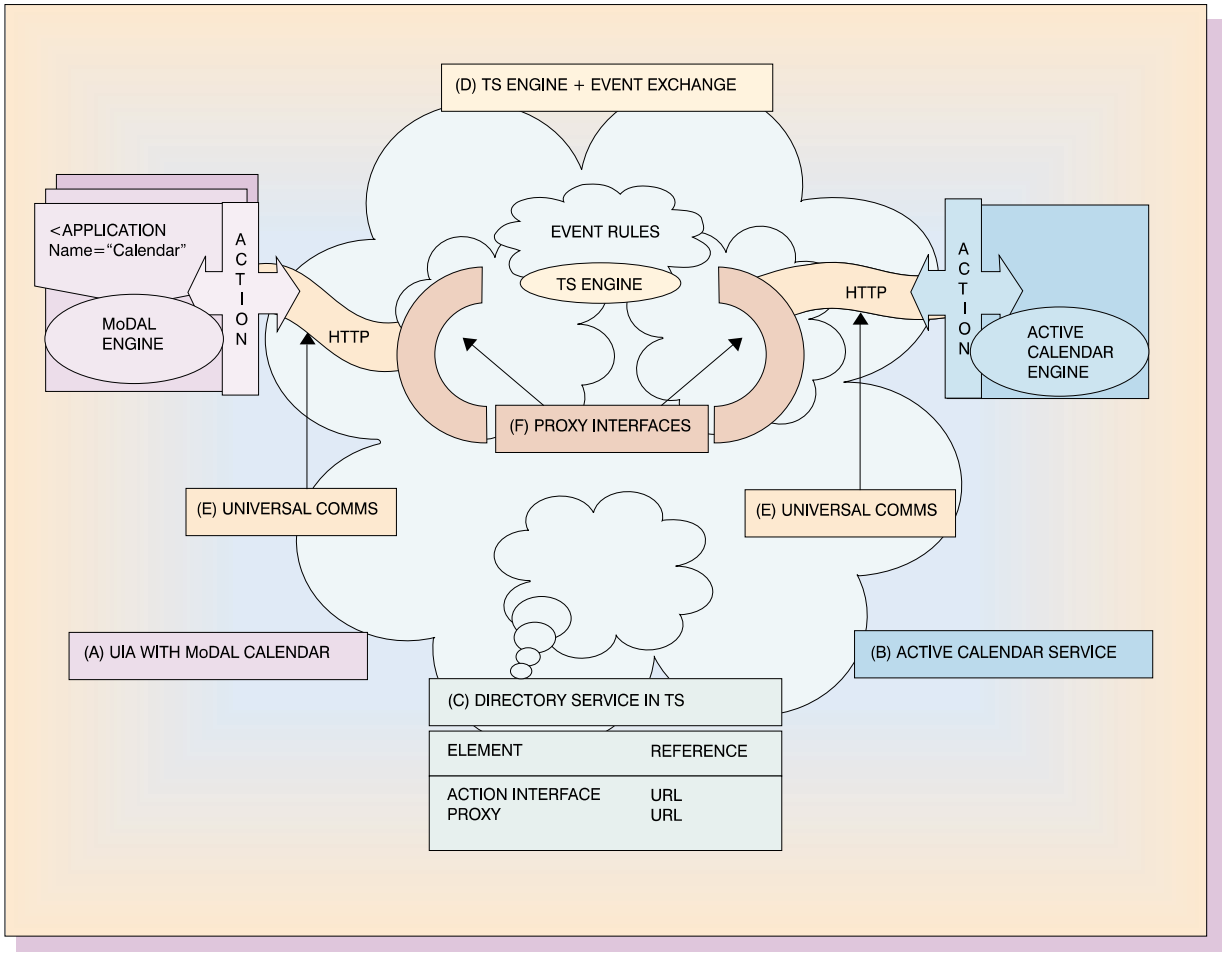
An example—the UIA active calendar. The principles outlined in the previous subsection are our framework for building pervasive "push" applications of many varieties. For example, consider an active calendar application extended for the UIA. The application includes a UIA calendar application (UIA), written in MoDAL, interoperating with the active calendar service (ACS) through TSpaces. When the user records a new calendar entry, the active calendar service sets out to find the pertinent information and downloads it to the client to display in the calendar application. For example, the UIA user might record an entry for a meeting with "John D" at company "XYZ-Soft" about "Project X"; the user prefers to fly into "LAX" airport. The ACS collects information for the meeting—the contact information for John D (e-mail address, cellular phone number, and job responsibilities) and the latest news on XYZ-Soft. The ACS also collects local information, including driving directions from LAX and local restaurants for a business lunch. The collected information is then downloaded through the TSpaces network and over the wireless link, to be displayed in the user's UIA calendar application.

Because the attributes of MoDAL elements are interpreted dynamically, both the style and the content of the UIA calendar interface can vary in real time. Thus, views can be tailored to the user's present needs ("Load driving directions when I am driving") and to the user's preferences ("Show Michelle a full company news report but omit news for Toby"). The interface can also be tailored to the hardware device capabilities. For example, on the PDA display only thumbnail images, "snippets" of local restaurant information, and text reminders appear, but the mobile PC, or the embedded auto-PC, display a full map, and play reminders as speech.

Active calendar components. The proposed active calendar system for the UIA would consist of the following components identified by the corresponding letters in Figure 9.

A. UIA client with MoDAL calendar application—The client publishes a generic action interface defining the MoDAL actions to be invoked in response to events in TSpaces. For example, an interface may declare a method that displays in a MoDAL UI object the contents of a tuple retrieved from TSpaces. The action interface is published to a well-known directory service.
B. Active calendar service—The ACS, also a client of TSpaces, publishes a generic action interface in XML. The interface defines action methods, exposing the internal ACS methods to TSpaces.
C. Directory service—The directory service is well-known and is used by the TS engine to locate appropriate action interfaces and by TSpaces clients (the UIA client and the ACS) to locate proxies for connecting to TSpaces. The contents of the directory service are stored in TSpaces, so that the directory service can be queried using a standard protocol to look up interface or proxy references by attribute.
D. Universal communication protocol—The UIA and application services, like active calendar, require a universal protocol to communicate with TSpaces. HTTP is a possible choice for several reasons. Most TSpaces clients can conveniently execute a Web server, URL-based locators provide a natural global naming scheme, and HTTP is a simple and widely used protocol for markup-language communications, such as MoDAL programs and XML action interfaces.
E. TS engine with event exchange capability—The TS engine manages all event communication between the UIA client and the ACS according to event rules, which are parsed and executed dynamically. On interpreting each new rule, the TS engine registers with the appropriate tuple spaces. When notified of a new event, the TS engine searches the rule store to match the notified event. If a match is found, the TS engine performs a lookup in the directory service for the action interface method named in the rule. The TS engine connects to client(s) implementing the interface and invokes the designated action, or alternately, writes a tuple to the local tuple space of the client, which triggers the client to do the action.

Figure 9 UIA active calendar system components



F. Flexible proxy interface to TSpaces—If HTTP is assumed to be the universal communication protocol, a transcoding proxy, such as IBM's WBI technology, translates between the client's HTTP request and TSpaces. On the upstream (client to TSpaces) connection, the proxy converts a client's HTTP post of a tuple space command (TSWrite, TSRead) to the corresponding TSpaces protocol. On the downstream (TSpaces to client), the proxy allows the TS engine to select the action interface of the client.

Active calendar architecture. Our approach for building an active calendar application for the UIA is summarized as follows:

- We open the UIA client (UIA) and the active calendar service (ACS) to exchange events over the network through TSpaces. The UIA and the ACS write events as tuples into TSpaces. Both expose their internal event-handling methods to events in TSpaces by publishing interfaces for "action injection."

- The "action injection" interface for the UIA calendar describes how the attributes and values of MoDAL elements, such as text displayed in textfields, will be set by events in TSpaces. For example, the calendar will implement an action method to display information written to TSpaces (by the ACS) in the relevant calendar entry.

- TSpaces manages the event communication between the UIA and the ACS, according to event rules as introduced above: Rule = TSClient.EventClass(parameters) → TSClient.ActionClass(parameters). When the TS engine interprets a new rule, it registers for the designated event with the appropriate tuple space on behalf of the client object, i.e., the UIA calendar or the active calendar service. When an event for which the TS engine has registered arrives in TSpaces, the TS engine is triggered to check its stored rules for a matching event template. If a match is found, the TS engine follows the rule and invokes the appropriate action interface of the client to be triggered. The event rules are object-oriented, which allows event parameters to be readily passed into the action invocations. Because interpretation is at run time, clients can be registered for new event classes without recompiling.

To illustrate these principles, let us consider the example active calendar scenario introduced previously. Step-by-step illustrations of the run-time flow are shown in Figures 10 and 11. You, an active calendar user, have an upcoming meeting with "John D" at company "XYZ-Soft" about "Project X." You will be flying via "LAX" airport and wish to have both driving directions to the meeting from the airport and suggestions of nearby restaurants for a business lunch. You have recorded these preferences in your UIA calendar application, which has uploaded corresponding event rules to the TSpaces engine. Now, when you store the new calendar entry (by, for example, pressing a "store" button in your UIA calendar application), the TSpaces middleware goes into motion on your behalf.

(Referring to Figure 10, we continue with the illustration.) Your entry is packaged into a tuple, which is sent over the wireless link to TSpaces (Step 1). The event management within TSpaces (Step 2) in turn invokes the active calendar service, also a client of TSpaces, to find a pertinent meeting and location information on your behalf (Step 3). (Referring to Figure 11, we see the remaining steps.) The active calendar service returns the pertinent information to TSpaces (Step 4). Again, the event management in TSpaces passes the information back to your calendar application (Step 5) and invokes the appropriate MoDAL action defined in the application interface to display the information in your UIA calendar in real time (Step 6).

Note that in this scenario, neither the UIA client nor the active calendar service needs to be known to each other, nor must they be in a single administrative domain or use the same TS server. Furthermore, because MoDAL interfaces are dynamically interpreted, the same active calendar service can be offered on the user's desktop PC, or in an auto-PC, or any platform with a MoDAL interpreter, for example. Because the interaction of the ACS and the UIA calendar application is decoupled from either client's internal event model and is wholly asynchronous, the same service can be offered to many heterogeneous calendar applications with information sent on an as-needed basis. For example, capability exists such that a user can make calendar entries on any MoDAL calendar application (independent of hardware and interface configuration) and receive the helpful information when appropriate to his or her context. For example, driving directions can be downloaded to the user as he or she is stepping out of a plane by accounting for the flight time, or even for the real-time location coordinates, in the TSpaces event management.

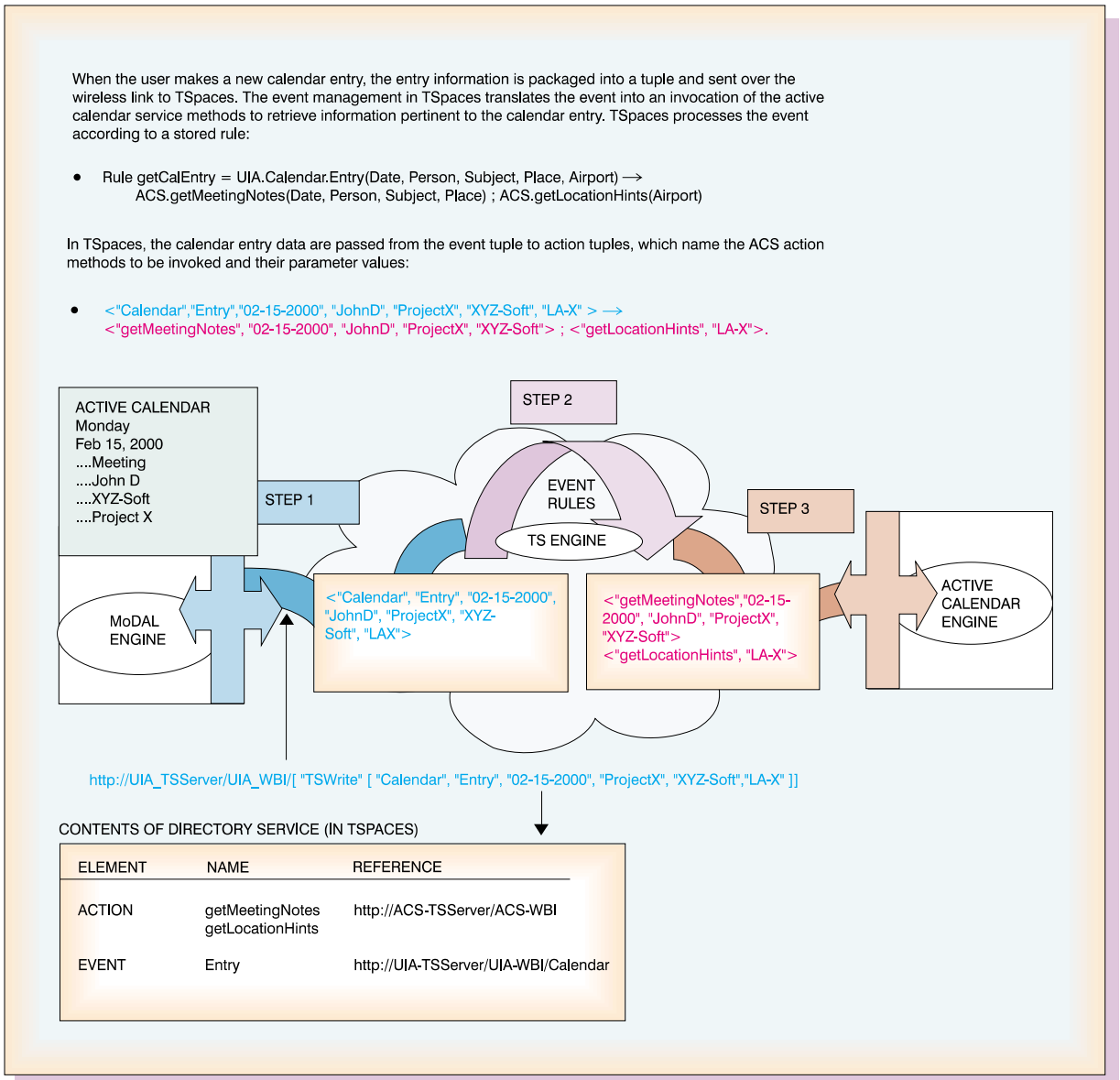
Related work

Several other projects have looked at various components of the system we have outlined here. None have focused on the system that we have presented here, namely XML application description and generation for wireless interaction with heterogeneous mobile devices.

The most pertinent of the related projects are the BARWAN and Ninja projects at the University of California, Berkeley. Hodes and Katz describe a system of static interface description and generation based on XML descriptions of device interfaces.[30] The focus of this work is to investigate interface generation and the customization of interfaces, both interesting and applicable to our work on the UIA, but not overlapping with our current UIA design of dynamic application-based interfaces.

Another related research project is the Rover project at the Massachusetts Institute of Technology.[31] The Rover project investigated queued remote procedure calls and relocatable dynamic objects, specifically focusing on the unique requirements of mobile environments. One aspect of the Rover project involved dynamic invocation of mobile applications. The Rover toolkit supports the retrieval and execution of static applications. The UIA differs from this by supporting dynamic application updates via the

Figure 10 Run-time flow for the UIA active calendar: invoking the ACS from the UIA



MoDAL interpreter and by utilizing a high-level interface language (XML), as well as extending the focus to remote device control.

Although the UIA model includes database support on the device and in the middleware, the dynamic nature of the MoDAL engine results from its combination of features (dynamic compilation, wireless communication to network middleware, local computation, and on-board cache). Thus, the few database platforms now available for PDAs do not compete. Sybase's SQL-Anywhere** product[32] is a fully functional SQL (structured query language) database system that runs on multiple PDA platforms. However, it is only a database system. Applications that use SQL-Anywhere are statically compiled, loaded,

Figure 11 Run-time flow for the UIA active calendar: pushing information to the UIA

When the active calendar service has located helpful information for the calendar entry, it packages the information in a tuple and writes the tuple to a local (well-known) tuple space. The event management in TSpaces translates this "event" into the injection of a MoDAL action, according to stored event rules:

- Rule setMeetingNotes = ACS.Meeting(Date, Person, Email, Cell, Position, News)
  → (UIA.Calendar.getEntryforDate(Date)).setNotes(Person, Email, Cell, Job, News)

- Rule setLocationHints = ACS.Location (Airport, Directions, Restaurants)
  → (UIA.Calendar.getEntryforAirport(Airport)).setLocationHints(Directions, Restaurants)

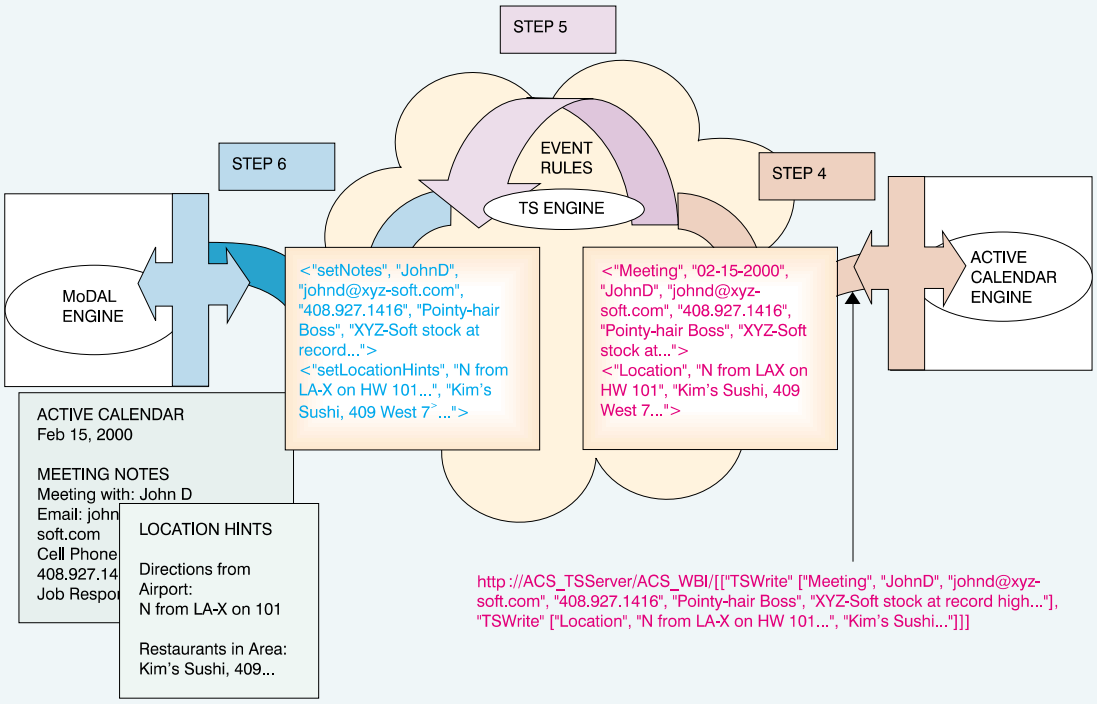
The user sees the information gathered by ACS displayed in his or her UIA calandar.

The event management in TSpaces causes the ACS information to be passed from event tuples to action tuples, which name the MoDAL actions to be injected, including parameter values. When the action tuples are written to a tuple space local (well-known) to the proxy of the UIA, the proxy is notified and "pushes" the actions to the UIA calandar. Alternatively, in a pull approach, the action tuples persist in the space until the UIA queries for new actions.

Example event and action tuples for the user situation shown in Figure 10 are given below:

- <"Meeting", "02-15-2000", "JohnD", "johnd@xyz-soft.com", "408.927.1416", "Pointy-hair Boss", "XYZ-Soft stock at record high..."> → <<"Calendar", "getEntryforDate", <"02-15-2000">>,"setNotes", <"JohnD", "johnd@xyz-soft.com", "408.927.1416", "Pointy-hair Boss", "XYZ-Soft stock at record high...">>

- <"Location", "LA-X", "N from LAX on HW 101...", "Kim's Sushi, 409 West 7..."> → <<"Calendar", "getEntryforAirport", <"LA-X">>, "setLocationHints", <"N from LAX on HW 101...", "Kim's Sushi, 409 West 7 ...">>



and run, in the same manner as in any other static-programming environment.

There are also several projects relating to our actual implementation of the UIA on the 3Com PalmPilot (or IBM WorkPad). The first project is Gary Desrosiers' dynamic user interface creation on the PalmPilot.[13] Desrosiers showed how to create a library of user interface routines that could be called to invoke user interface components dynamically. We extended that notion with a full language description (MoDAL) and a run-time interpreter for MoDAL that reads the MoDAL language, builds the user interface, and integrates local database support and a TSpaces network interface. A competing project at the IBM Almaden Research Center tried to achieve similar dynamic UI functionality. Daniel Kellem determined the in-memory structures that are created by the Palm OS when it transverses the data structure for the forms resources and wrote several routines to construct those resources via API calls. Kellem's approach is somewhat more dynamic and finer-grained than Desrosiers'; for example, Kellem's approach makes it possible to add a button to a form already displayed. However, this simple UI work became mostly obsolete with release of Palm OS 3.0, which provides an API for dynamically generating forms.

Also related to our implementation are a number of forms building tools, of which Softmagic's Satellite Forms** [33] is an example. Satellite Forms is a software product for developing user interfaces. It allows application developers to use their UI-based tool to create static applications for the IBM WorkPad. In addition, these programs can interact with back-end programs, such as Oracle databases and Lotus Notes.

The forms building tools have not evolved beyond pure user interface functions. These tools do not employ techniques for dynamic interface description, nor do they incorporate an integrated client database function, nor do they feature a connection to a general-purpose software switchboard, such as TSpaces. The current method for creating PDA applications is based on the old development cycle of edit, compile, and test run. The compile step is the one that, for all practical purposes, prevents the current generation of PDA user interface tools from generating dynamic interfaces—the descriptions simply cannot be changed "on the fly."

Numerous tools are available for generating dynamic HTML (HyperText Markup Language) documents, documents in which style and content can be adjusted "on the fly." However, none of these tools is a substitute for MoDAL or the MoDAL engine. We will explain why by first explaining what Dynamic HTML is and then explaining how it differs from the MoDAL programming model.

Dynamic HTML is an amalgam of proprietary and standardized technologies for adding dynamic behavior to HTML documents, driven mainly by Microsoft's Internet Explorer** Web browser and Netscape's Navigator**.[34] Dynamic HTML technologies include scripting languages, a document object model (DOM) that allows scripts to be applied to document objects, and style sheets. Within these three broad areas are numerous competing technologies. For example, scripting languages include Netscape's JavaScript**, Microsoft's VBScript** (derived from Visual Basic**) and JScript** (compatible with Netscape's JavaScript), and the politically neutral script ECMAScript, which unifies JavaScript and JScript (as of version 4.0 of both Netscape Navigator and Internet Explorer).

As of version 4.0, both Navigator and Internet Explorer allow HTML document content, attributes of tags and styles, and document object position to be dynamically set when a new document loads, through executable script, which triggers on the load event. Internet Explorer 4.0 goes a step further, in that document content for a *loaded* page can be dynamic, triggering in response to data transfer to the client or user events. (Valid events are, for example, a mouse click, selection or focus on a document element, a window resizing, a keyboard press, or the interruption of a document load). For example, consider a news article filter script, which searches for and prioritizes stories containing keywords selected by the user. When the user selects a new keyword by, for example, clicking on a checkbox in a browser window, the script is launched. The script clears the current articles, searches and prioritizes articles containing the keyword, and finally reloads the page displaying the new digest of matching articles.

An important distinction between the dynamic content provided by Dynamic HTML and the MoDAL programming model is the connection to general-purpose middleware (TSpaces). As shown previously in the active calendar example, MoDAL document content and element attributes are dynamic to events and data in TSpaces. In principle, *any TSpaces client*—any application service, database, context-sensor, or device that implements the simple TSpaces

client API or sends messages to a proxy—can trigger the content or style of a MoDAL application. This is significantly more powerful than the Dynamic HTML model where the dynamic behavior of document objects is limited to events internal to the browser client, or generated by the Web server. For example, Internet Explorer does include event handlers that pertain to form elements bound to server database sources. Java servlets[35] allow client-side applets to execute code on the Web server, such as code to query an SQL database, to access file service, news, or chat services over HTTP, or to submit form information to the server and receive dynamically generated HTML. However, the Web server is not general-purpose middleware, implicitly open to any data or event source in the network. The Web server interacts only with known services that implement specific interfaces and comply with a specific event model. Because TSpaces supports asynchronous, anonymous communication and has an event engine to bridge between specific event models, TSpaces allows the MoDAL client to be a generic network client to potentially any service.

We do believe that the various Dynamic HTML specifications are a useful starting point for extending Web browsers with MoDAL-style connectivity to TSpaces. We envision a world in which every user device has the capability to dynamically generate application interfaces, which have generalized communication with the network infrastructure. The current generation of Web browsers could be extended, for example, with event handlers that are called as a result of events in TSpaces or with primitives to send messages to and query TSpaces. Our first step toward unifying MoDAL and Dynamic HTML is developing an applet that communicates with TSpaces. A next step may be to drop the functionality of the applet into the Web browser itself, for example, with a MoDAL plug-in. On the MoDAL side, we plan to research how the MoDAL language can be made compatible with Dynamic HTML. Although there are no browsers for the IBM WorkPad, Microsoft's WinCE** operating system does run a limited version of Internet Explorer. Thus, to avoid duplication, it will be important to understand how MoDAL capabilities can be integrated with the browser.

Concluding remarks

The end goal of the UIA project is to create a device sufficiently flexible to incorporate any new client application or interface that is exported by a server application or electronic device. Furthermore, through a connection to TSpaces, wireless or otherwise, the UIA has enormous power for triggering external events, as virtually any service can be brokered by TSpaces. We have created the MoDAL language, a high-level XML-based description language that can represent the application user interface, local scripting-like computation, local database actions, and remote network messages to TSpaces. With MoDAL, applications and devices can create relatively advanced client programs with which a user can interact. With this as a platform, we can raise the level of human-machine interaction. By exploiting the dynamic interface capability, applications (both new and legacy) can tailor the user interface to fit the user's context, background, experience, and immediate needs. Furthermore, the current trend of multiple remote controls and extremely complex controls will eventually give way to single universal soft remote control devices.

By achieving our goal, we not only advance the science of human-machine interaction, but we change the way in which we are able to look at the world of information. By having a constant connection to any data desired, the number of mistakes the average person makes in a day will decrease (lost names, lost phone numbers, lost directions, missed appointments, missed events). More fundamentally, a user's opportunity to simultaneously experience many different contexts of life is extended because the UIA augments one's senses, hands, and "curiosity." The UIA enables us as users to engage in real-world pursuits—to discover information, manipulate devices, and collaborate with other people—as if we were not bounded by real-world realities, such as separations in geography, time, or context. Although this achievement does increase our dependence upon the world's information servers, it also potentially enhances our quality of life.

The TSpaces project released version 2.0 to the public via the IBM alphaWorks* channel[36] on October 30, 1998. The release contained the full TSpaces system and the beginnings of the PDA support, though the generalized MoDAL engine and the wireless support are still under development. The MoDAL language is undergoing its first revision, because several other internal IBM groups have come forward, expressing the desire to merge their XML-based functions with MoDAL to create a unified and more expressive language that can serve many purposes.

In addition to opening the MoDAL/TSpaces event model, we plan to involve the MoDAL specification

Figure 12 Code for "Credit Me" application

```
<APPLICATION name="PilotPop">
<FORM name="Main" title="PilotPop" x="0" y="0" width="160" height="160">
        <BUTTON name="credit" label="Credit Me!" X="65" y="20" width="30" height="15">
                <ACTION>
                        <TUPLE space="POPcommand" server="TSpaces">
                        B2_253Machine
                                <AND>CREDIT</AND>
                                        <VAL>0<VAL>
                                </AND>
                        <TUPLE>
                 <ACTION>
        <BUTTON>
</FORM>
</APPLICATION>
```

in standardizing the next-generation UI languages. MoDAL is well-positioned to influence standards for the semantic representation of data and events for sharing among heterogeneous clients. For example, the current implementation of the active calendar taps various information sources to generate standardized XML metadata descriptions; these descriptions coincide with calendar entries to select the information to be pulled by the calendar client. We plan to explore how such metadata descriptions can be unified with the syntax for describing MoDAL elements. For example, when an upcoming meeting event at the Almaden Research Center is recorded in an active calendar client, TSpaces might match up MoDAL elements tagged with <LOCATION = "IBM Almaden Research Center"></LOCATION> to metadata about the Almaden Research Center published on its Web server, in order to deliver driving directions to the traveler en route.

The MoDAL engine for the IBM WorkPad is one of what we expect to be many implementation flavors of the UIA. The next version of the UIA may be a replacement for the present Web browser, which will allow the mobile ThinkPad*, for example, to have soft interfaces. One use of this general-purpose information presentation is the "world board" concept,[37] whereby mobile users are continuously presented new information and new interfaces that are associated with the user's physical locations. We expect that subsequent versions of the UIA will be driven by the development of a demonstration integrating all of the UIA components—the MoDAL engine, the short-range (Bluetooth) and campus-range RF wireless connection, the TSpaces engine with a lightweight event-processing service, and existing pervasive "push" applications, such as the active calendar. By building a set of pervasive applications using MoDAL as the application scripting language, TSpaces as the middleware, and wirelessly connected mobile user devices as the hardware platform, we aim to better understand and address the practical obstacles to the UIA vision.

Acknowledgments

The authors thank the TSpaces team, Dwayne Nelson, Keung Hae Lee, John Thomas, and Cal Leister for their valuable comments and contributions. We extend a special thanks to Dwayne for his artistic contributions. We also thank Norm Pass for his enthusiastic support of our project and the reviewers for their thoughtful comments and suggestions.

Appendix A

The MoDAL code for the "Credit Me!" (or "PilotPop") application is shown in Figure 12.

Appendix B

The members of the interface elements resource are:

- *resourceID*: The resourceID serves as an identifier for graphical elements. Every graphical component will have an identifier, unique to the interface. Non-UI form components inherit the resourceID of their parent form, with the exception of ACTION elements, which have a resourceID of 0.
- *token*: This flag stores the element type, i.e., FORM, BUTTON, ACTION, etc., to allow the engine

to quickly determine if an element is a graphical element, or an event-handler.

- *valueType*: For components of type ELEMENT, this can be any valid MoDAL type. For all other components, this is 0.
- *value*: For UI components, this flag stores the name of the associated UI element. If the component is of type ELEMENT, data of the type valueType is stored in value. Finally, if the component is an attribute of a graphical component, or an action, value refers to the component to be operated on, i.e., where to retrieve data from, copy to, etc.
- *next*: The various interface components can be viewed as a linked list, created at parsing. This field is a pointer to the next component in the interface. This pointer can be used to poll the various components for events.
- *attr*: This field is a pointer to the next child component, or attribute. The various interface components contain attributes describing and modifying their behavior. In lieu of an attribute, the component may possess child components.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Starfish Software, 3Com Corporation, Sun Microsystems, Inc., Lotus Development Corporation, Microsoft Corporation, Sybase Corp., Softmagic Corp., or Netscape Communications Corp.

Cited references and notes

1. The universal information appliance effort is a part of the TSpaces project.
2. We use the term "electronic systems" to mean anything electronic, including computers and the programs running in them.
3. The Pronto Universal Remote is listed at http://www.mmhometheatre.com/components/rvpronto.html, and the Marantz Mark II Universal Remote is listed at http://www.marantzamerica.com/rc-2000.htm.
4. The Blue Mountain Network Computer running Aplix Jblend is described at http://jblend.com/product/p_jblend.html.
5. We considered an olfactory interface, but then gave up when the thought became too unsavory.
6. The IBM Blue Eyes Project is described at http://www.almaden.ibm.com/cs/blueeyes.
7. The Rex Pro PDA, by Starfish Software, is found at http://www.starfish.com/products/truetech/index.html.
8. T. G. Zimmerman, "Wireless Networked Digital Devices: A New Paradigm for Computing and Communication," *IBM Systems Journal* **38**, No. 4, 566–574 (1999, this issue).
9. Such as Bluetooth (http://www.bluetooth.com/) wireless links or IEEE 802.11 wireless LAN systems.
10. "Push!," *Wired Magazine* **5**, No. 3, 12–23 (March 1997).
11. The word "document" refers not only to traditional documents, like this one, but also to innumerable other forms of XML "data formats," including mathematical equations, database schema, client/server APIs, and vector graphics, as well as many other forms of structured data.
12. XML.COM at http://www.xml.com/xml/pub.
13. G. Desrosiers, "Dynamic User Interface," http://www.vermontlife.com/gary/DynamUI.html.
14. A. Morales and M. Guillen, "The MoDAL Language Specification," IBM Working Document.
15. P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford, "T Spaces," *IBM Systems Journal* **37**, No. 3, 454–474 (1998).
16. The data management portion of TSpaces was modeled after the Starburst main memory data manager. See T. J. Lehman, E. J. Shekita, and L.-F. Cabrera. "An Evaluation of Starburst's Memory Resident Storage Component," *Transactions on Knowledge and Data Engineering* **4**, No. 6, 555–566 (1992).
17. Tuplespace is a concept created by the Linda project at Yale University. See: D. Gelernter and A. J. Bernstein, "Distributed Communication via Global Buffer," *Proceedings of the ACM Principles of Distributed Computing Conference* (1982), pp. 10–18; D. Gelernter, "Generative Communication in Linda," *TOPLAS* **7**, No. 1, 80–112 (1985); N. Carriero and D. Gelernter, "Linda in Context," *Communications of the ACM* **32**, No. 4 (April 1989).
18. A tuple is the basic carrier of data in a Tuplespace. A tuple is merely a vector of typed values.
19. T. A. Howes and M. C. Smith, *LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol*, MacMillan Publishing, New York (1997).
20. Service Location Protocol is found at http://www.swrloc.org/index.html.
21. Salutation is at http://www.salutation.org.
22. Jini specification is at http://www.java.sun.com/products/jini.
23. The IBM WorkPad specification is at http://www.pc.ibm.com/us/workpad.
24. Information about Motorola is at http://www.motorola.com.
25. Q. Lu, S. Edlund, D. Ford, and U. Manber, "Active Calendars," submitted for publication, IBM Almaden Research Center (1998).
26. We call micro-sized RF transmitters broadcasting to passive RF receivers, "RF Bugs." The "Bug" technology is currently under development at the IBM Almaden Research Center, under the direction of Tom Zimmerman.
27. M. Munson, "System Support for Composing Distributed Applications Using Events," dissertation for Diploma in Computer Science, University of Cambridge, UK (August 1998).
28. J. Bates, M. Spiteri, J. Bacon, and D. Halls, "Integrating Real-World and Computer-Supported Collaboration in the Presence of Mobility," *Proceedings of WETICE'98* (1998).
29. IBM Web Browser Intelligence (WBI) is at http://www.almaden.ibm.com/cs/user/wbi/index.html.
30. T. D. Hodes and R. H. Katz, "Enabling 'Smart Spaces' Entity Description and User Interface Generation for a Heterogeneous Component-Based Distributed System," *DARPMNLST Smart Spaces Workshop*, Gaithersburg, MD (July 1998), University of California, Berkeley Technical Report CSD-98-1008.
31. The Rover Web site is at http://www.pdos.lcs.mit.edu/rover/.
32. Information about Sybase is at http://www.sybase.com.
33. Information about Softmagic is at http://www.softmagic.com.
34. D. Goodman, *Dynamic HTML: The Definitive Reference*, O'Reilly & Associates, Sebastopol, CA (1998).
35. Sun Microsystems, "Java Web Server" can be found at http://www.sun.com/software/jwebserver/features/index.html.
36. The URL for alphaWorks is http://www.alphaworks.ibm.com.
37. J. Spohrer, "Information in Places," *IBM Systems Journal* **38**, No. 4, 602–628 (1999, this issue).

General references

N. Carriero and D. Gelernter, "Linda in Context," *Communications of the ACM* **32**, No. 4, 444–458 (April 1989).

D. Gelernter, "Generative Communication in Linda," *TOPLAS* **7**, No. 1, 80–112 (1985).

D. Gelernter and A. J. Bernstein, "Distributed Communication via Global Buffer," *Proceedings of the ACM Principles of Distributed Computing Conference* (1982), pp. 10–18.

JavaSpace specification, http://www.java.sun.com/products/javaspaces.

T. J. Lehman, E. J. Shekita, L.-F. Cabrera, "An Evaluation of Starburst's Memory Resident Storage Component," *Transactions on Knowledge and Data Engineering* **4**, No. 6, 555–566 (1992).

Accepted for publication March 26, 1999.

Kevin F. Eustice *13768 Trost Trail, Savage, Minnesota 55378 (electronic mail: kfe@cs.hmc.edu).* Mr. Eustice is pursuing a Ph.D. degree at the University of California, Los Angeles. He graduated in May 1999 with a B.S. degree in computer science from Harvey Mudd College. He has been involved with the TSpaces project at IBM Almaden since May 1998. His research interests include mobile computing, distributed systems, and computer-human interaction, as well as caching and replication in mobile databases. His goal is to make the world fully connected and fully interoperable using mobile and distributed systems.

Tobin (Toby) J. Lehman *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099 (electronic mail: toby@almaden.ibm.com).* Dr. Lehman joined the IBM Almaden Research Center in 1986, shortly after finishing his Ph.D. degree from the University of Wisconsin-Madison. He is currently a member of the Computer Sciences Division. His research interests include server-based backup systems, object-relation database systems, large-object management, memory-resident database systems, Tuplespace systems, and just about anything written in Java. Dr. Lehman is the leader of the TSpaces project, and he plans to use TSpaces to make the world a more productive place.

Armando Morales G. *IBM Mexico, Guadalajara Software Group, Carretera el Castillo Km 2.2, El Salto, Jalisco, Mexico 45550 (electronic mail: armando@mx1.ibm.com).* Mr. Morales graduated as an electrical engineer in 1984. He joined IBM in 1985 as a product engineer for the System/36™ and later for the AS/400®. In 1990 he finished an M.B.A. program, and in 1992 he joined the Guadalajara Programming Laboratory, where he has been a team leader for multiple software projects for the AS/400 and lately for Netfinity® servers. He is interested in new applications for small devices such as the IBM WorkPad.

Michelle C. Munson *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099 (electronic mail: munsonm@almaden.ibm.com).* Ms. Munson joined IBM and the TSpaces project in 1998, after finishing a master's degree in computer science at the University of Cambridge in the United Kingdom. Currently a member of the Computer Sciences Division, her interests include all aspects of mobile computing and distributed object systems. She is interested in technologies that interact with the physical and electronic domains for context-aware and augmented reality computing. She graduated with B.Sc. degrees in electrical engineering and physics from Kansas State University in 1996. She enjoys running, meditation, and, most recently, gambling.

Stefan Edlund *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099 (electronic mail: edlund@almaden. ibm.com).* Mr. Edlund graduated with a master's degree in computer science from the Royal Institute of Technology, Stockholm, in 1997. Since then, he has worked at the IBM Almaden Research Center developing applications for the DB2® database system, and he has more recently done work on various Web applications, among them the active calendar. Currently a member of the Computer Sciences Division, his primary interests include the Web, Java, PDAs, data mining, and playing music.

Miguel Guillen G. *IBM Mexico, Guadalajara Software Group, Carretera el Castillo Km 2.2, El Salto, Jalisco, Mexico 45550 (electronic mail: mkt@mx1.ibm.com).* Mr. Guillen joined IBM in 1990, after finishing his electrical engineering studies at the University of Guadalajara. In 1992, he joined the Guadalajara Programming Laboratory, where he has been developing software for different projects including the AS/400, Netfinity, and the Advanced System Management Adapter. His main interests are little devices, such as PDAs and microcontrollers, object-oriented design patterns, teaching C++, using high-end audio, and cooking.

Reprint Order No. G321-5707.