System Support for Interactive Workspaces

26th March 2001

Abstract

While work has been done on connectivity, mobility and general rendezvous systems for ubiquitous computing environments, research has been hampered by the lack of higher level application abstractions. To identify what these might be, we constructed a prototype ubiquitous computing environment which we call an Interactive Workspace, and observed how application writers attempted to use its facilities. We also identify system support issues for ubiquitous computing that differ from their counterparts in "traditional" single-node computing, and use them to guide the design of a set of abstractions to support ubiquitous computing applications deployed in spaces such as ours. Our implemented prototype meta-operating-system, iROS, and several applications in daily use on top of it, suggest that we have made progress toward the goal of providing support for application-level ubiquitous computing abstractions in a way that is robust to transient failures, extensible, portable across installations, and easy to program. Finally, as we consider future work, we argue that the abstractions we identified—moving data around, moving interfaces around, and coordinating the behavior of existing monolithic applications—are fundamental to this style of ubiquitous computing generally.

1 Introduction

Improvements in device technologies and falling costs are rapidly enabling the original vision of ubiquitous computing [29]. Devices from large wall-sized displays to small PDAs can easily (and wirelessly) be networked together in localized areas, forming the hardware side of the ubiquitous computing environment. Once connected together, however, these devices do not generally integrate well with one another or with existing software unless they were designed to do so a priori. The ad-hoc interactions and applications envisioned for ubiquitous computing are difficult to achieve because there are no higher-level system abstractions for allowing this collection of off-the-shelf components to interoperate. Worse, such systems must be able to tolerate a dynamic environment, as portable devices come and go, as well as maintain a high degree of robustness and availability despite inevitable software and hardware failures. We address this gap with the following contributions:

- A discussion of systems support for ubiquitous computing and how it differs from systems support for "traditional" single-node computing
- The identification of fundamental abstractions for application developers in room-based ubiquitous computing, based on our experience building a prototype space and inviting researchers to write applications for it
- A system model and implemented meta-operating-system that provides support for the high-level abstractions we repeatedly observed in ad-hoc applications, while remaining robust, extensible (new services and devices are easy to integrate), and easily portable to other installations and to devices with other operating systems
- Our experience using this software on a daily basis and deploying it to other sites with similar interests.

The meta-operating system is designed to supplement, not replace, the standard operating system facilities provided by individual machines. The paper proceeds as follows. We describe how we built a prototype room to learn what abstractions application developers would require. We then distinguish the system support requirements of ubiquitous computing from those of "traditional" computing, and identify the mechanisms necessary to provide these abstractions while satisfying the systems constraints. We describe and evaluate our implemented meta-OS and various applications running on it that are in daily use. Finally, we argue that given our observations and experience, the abstractions and system model we propose for this style of ubiquitous computing are fundamental, beyond our specific prototype implementation.

2 Characterizing System Support for Interactive Workspaces

We began by building a prototype interactive workspace, the iRoom. As can be seen in figure 2, it contains a variety of display devices and facilities for wireless and wired networking

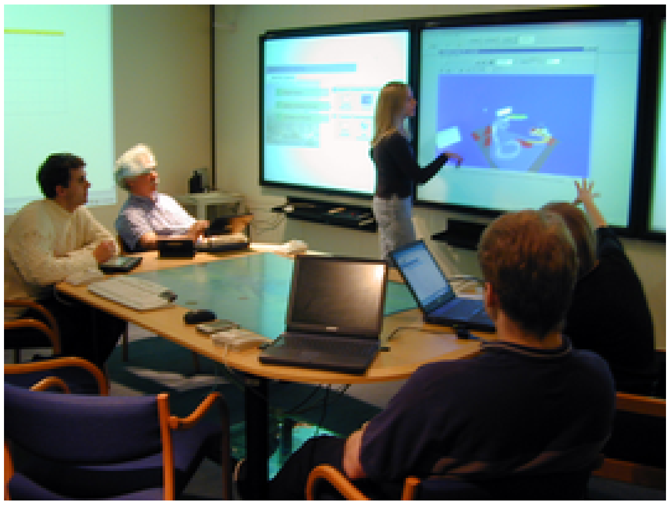

Figure 1: The iRoom contains the Mural [4], a high-resolution wall display addressable only in OpenGL; three SmartBoards, off-the-shelf low-resolution touch-sensitive wall displays powered by Windows 2000 PC's; a tabletop display; and wired and wireless network support for portable devices.

of occupants' portable devices, and we argue that it is representative of an important class of ubiquitous computing installations.

To understand how such a space would actually be used, we invited research groups from within computer science and other departments to use the iRoom and collaborate with us on developing applications for it. A typical usage scenario ran as follows. A construction management team comes to the iRoom for a planning meeting about a new project. One team member turns on the room lights and projectors with a user interface she pulls to her PDA. Some engineers then move 3D models of the construction site from their laptops into a common room dataspace, and the meeting manager directs several different views of the model onto the screens in the room. Other members view the models on their laptops or PDA's, but since the model is too complex it is automatically displayed as a set of rendered images showing some key views of the model.

The financial planners are concerned that the new plan is too expensive, so they replace a 3D model on one of the screens with a financial model using their PDA to choose the application and screen. Since the financial data and 3D model are already tagged with similar object names, the meeting manager directs the room to associate the financial model with the remaining 3D views, and as various objects in the 3D model are selected the meeting members are able to observe the related financial data.

2.1 Application Developer Facilities

The scenario above is representative of the many that we observed or prototyped to varying degrees. From this process we identified three common modalities that emerged repeatedly and deserved to be supported as higher-level facilities for application writers:

A1. Data Movement Facility: A standard location and type independent data storage for the interactive workspace which is easily accessible to application developers. Design issues include standard distributed file systems problems, as well as transforming information into data formats understandable by and appropriate for a variety of client devices.

A2. Interface Movement Facility: A standard way to specify how applications can be controlled by users and other applications. The design issues here are related to work in user interface markup languages and service description and discovery.

A3. Dynamic Application Coordination Facility: A facility to allow any interactive workspace application to interact with others, including those not originally designed to be compatible.

2.2 Systems Support Issues

Compared to traditional single-node OS's, a "meta-OS" to tie together multiple devices in scenarios such as the above introduces some new constraints and qualitatively magnifies some existing systems constraints. We now describe these with their implications.

Heterogeneity and adaptation. There is no single dominant OS or programming environment in our scenario. OS demands may be dictated by various factors: the experience of programmers, the availability of device drivers for experimental I/O devices, the quirks of homebuilt equipment such as our hi-res Mural (which is addressable only in OpenGL). Therefore we must make it easy to adapt legacy software, hardware and OS's to our system, whether or not source code is available. This implies that we should use existing OS's and applications as large building blocks, and identify ways to use those applications potentially in ways the designer never intended, for example by "puppeteering" [7] the existing applications.

Further, the inherent heterogeneity of ubiquitous computing, and the introduction of new hardware over time, far exceeds what we are used to; approaches such as adaptation by proxy [15], which work well for adapting a variety of clients to servers, do not readily generalize to adapt clients to communicating with each other. Since the "least common denominator" functionality across clients is small, the API's and meta-OS code must be accessible from many languages and platforms, and they must be easy to port to new languages and platforms.

Finally, extensibility must be construed to include the integration of future hardware and software, which implies that we should strive to minimize the requirements and assumptions made about new hardware and software and minimize the amount of work required to integrate a new devices. The Internet has successfully addressed a similar issue by moving computation and services into the infrastructure rather than to client and server endpoints, in order to keep clients and servers simple; we advocate an analogous approach here.

Robust to Transient Failure. A system of dozens or hundreds of devices running hundreds of independent processes *must* be able to survive transient failures of any component; in fact, ideally it should be modularly restartable [12], so that in general the restart of any component is considered part of normal system operation.

Portable across installations. Just as applications written to run on a PC under some operating system are portable to other PCs, applications written to our infrastructure should be usable in other interactive workspaces with minimal or no changes. Applications should not be required to make assumptions about which specific components are present or absent, or about the geometric arrangements of those components in a particular installation. This implies that we should provide a way to structure "applications" as loosely coupled ensembles of autonomous components, and provide indirection mechanisms for addressing the components that are as independent of location as possible.

3 A Meta-OS System Model

We have developed a system model and implemented prototype that provide the above functionality subject to the systems support constraints we outlined. The foundation of the model is a communication infrastructure that meets the requirements given in the last section and the goals just given while facilitating dynamic application coordination. We then layered on top additional systems that use the communication infrastructure to provide data movement and transformation (A1) and interface generation and movement (A2) (shown in figure 2).

3.1 Communication Infrastructure

Our system is built on a tuplespace communication model. A tuple is an ordered collection of named, typed fields. Tuple sources deposit tuples into a logically centralized tuplespace; receivers query the space by specifying a match template constraining the attributes of interest. In addition to providing a layer of indirection between sender and receiver in a similar fashion to publish-subscribe, the tuplespace additionally decouples the communication in time since tuples can persist in the tuplespace. Thus, sender and receiver need not be active at the same time.

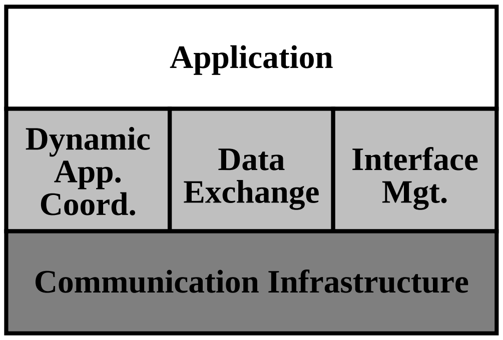

Figure 2: System Model of Infrastructure

Many of the tradeoffs between message-passing, RMI/RPC, publish-subscribe, and tuplespaces are well-known in the systems literature. Our contribution consists of evaluating these mechanisms relative to our problem domain. The tuplespace model has only four primitive operations, so it is as simple as any other technique for a client to support. Operations such as blocking RPC can be easily implemented on top of it, but by default, tuplespaces decouple senders and receivers in time as well as space, so that entities recovering from a failure can retrieve communications that were transmitted while they were down, so it affords potentially robust operation. This property is not present in publish-subscribe systems, though it is present in some distributed state management systems such as scalable reliable multicast (SRM) [11]. Both tuplespace and publish-subscribe models naturally support multicast communication between disparate groups over the same medium, which is more difficult with message passing or RMI/RPC; but publish-subscribe models are typically "receiver driven" in that events are usually not posted unless there are subscribers interested in receiving them. Because we do not necessarily know in advance whether a receiver will be interested in a particular post, we opt for the tuplespace model for this reason, as well as because the persistence of tuples can be bounded with an expiration time after which they are garbage collected if not consumed. This latter property also improves robustness since it eliminates the resource reclamation issue associated with having no receivers.

Tuplespaces do not scale well since all publishers and subscribers communicate through the same shared medium. However, a single workspace will contain perhaps tens of users and hundreds of machines or processes, so this is not a concern. There is also a potential performance pitfall since all communication is indirect, but we are concerned only that end-to-end latencies be below the human delay-perception threshold. From these observations, we concluded (and our experience has largely confirmed) that the tuplespace model works as well as or better than other communication techniques given our problem domain. Later we will argue, in light of our experience, that some kind of *logically centralized* multicast-like communication is actually fundamental to this style of ubiquitous computing.

3.2 Data Exchange System

In an interactive workspace, users' data needs to be easily accessible from any application for manipulation and display. However, the heterogeneity of the applications (and the devices they run on) in this environment makes it difficult for entities to directly share data with one another because they often do not understand the same data formats. In addition, the data sources in this environment are also heterogeneous, ranging from traditional file servers to live, streaming digital cameras.

To successfully support data exchanges among this variety of applications and data sources, the data exchange system has to provide a high-level decoupling between applications, and between applications and data sources, by providing both type-independence of data, and location-independence of data. That is, an application wishing to retrieve information should not be dependent on either the original data-type of the information, or the kind of device the information is located on.

To provide type-independence, the system must be prepared to automatically transform data from its original format into a format known to the destination application. Though data transformation is difficult to do perfectly, there are a number of methods that achieve acceptable results [21, 24]). Similarly, to provide location-independence, the system must be prepared to provide some mediation mechanism from the transfer protocol presented by the data source to that required by the application.

By providing this type-independent and location-independent view of data stored in an interactive workspace, the data exchange system is removing the dependencies applications once had on one another—resulting in a high-level decoupling along these axes, analagous to the low-level decoupling tuplespaces provide for control and coordination communication in our model.

3.3 Interface Management System

The goal of the interface management system (IMS) is to enable accessing any service from any user appliance. We would of course like to support widely-deployed UI renderers such as HTML and WML browsers, but the IMS should allow for generation of interfaces for any modality, including standard GUI/WIMP interfaces, voice interfaces, and others. To keep clients simple and minimize assumptions about them, we provide infrastructure resources for UI generation and adaptation. Similarly, to make integration and prototyping easy, the IMS should provide for automatic interface generation to the degree possible, so that adding a new service (or a user appliance) does not force the developer to create interfaces for every new appliance or every service; but it should also allow creation of custom interfaces, ideally supporting the full spectrum from automatic to handcrafted. Finally, the IMS should allow generation of interfaces that are portable across workspaces, but still adapt themselves to the context of the local workspace.

Although recent work in the industry and research laboratories (e.g. Jini [2], Hodes et al. [20], UPNP [13], Roman et al. [22]) has addressed user interfaces in ubiquitous computing environments, these systems do not adequately address all the above-mentioned goals. Their architectures do not directly facilitate portable, yet context-adaptable UIs or infrastructure support for UI generation/adaptation. Also, the Hodes and Jini approaches do not address UIs for off-the-shelf browsers, the UPNP approach targets only web browsers, while [22] is GUI-centric.

To meet all the above design goals, we interpose a new infrastructure component, the *interface manager*, between the services and the user appliances. Services announce their descriptions to the interface manager and appliances request interfaces from it. The interface manager selects a pre-defined interface, adapts a generic interface, or generates a new interface automatically based on the service description, appliance, and the context information of the local workspace.

The decoupling of services and appliances made possible by the Interface Manager allows us to conveniently handle appliance customization, context adaptation, and migration of UI generation/ adaptation to the infrastructure. Thus, the interface manager enables loose coupling for user interfaces just as tuplespaces do for inter-application communication and the data exchange system does for data exchange.

3.4 Dynamic Application Coordination

In an interactive workspace, it is not usually known in advance which software components will be communicating. Message passing and RPC generally require advance agreement between communicating partners on argument marshalling and data types, constraining applications to interacting with other applications that understand the same interfaces that they do. In both cases, it is possible to create custom applications to bridge two applications that use different message formats or interfaces, but this can be cumbersome and requires an understanding of the workings of both applications to know what formats or interfaces they use. Both the tuplespace-model and publish-subscribe, however, share the following properties which make them well suited to dynamic application coordination:

Self-describing events: Events or tuples (called events for simplicity in the remainder) are self-describing, and always go through a shared medium. This makes it easy to pick up events and determine their function.

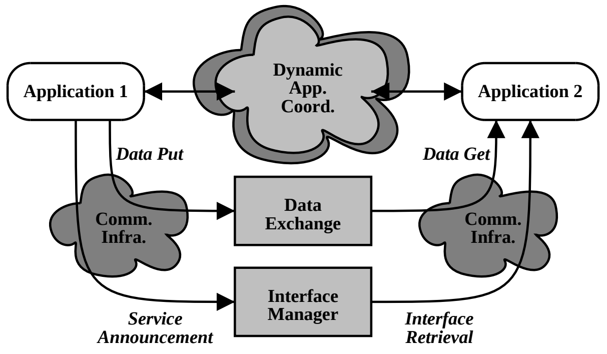

Figure 3: This system overview shows how IWork applications interact with each other through our communication infrastructure, coordination framework, data exchange facility, and interface manager.

Anonymous communication: Since there is no need to explicitly rendezvous applications, as long as two applications understand the same event types they will automatically coordinate with each other. This means users can bring up the applications they want where they want them and things should function correctly.

Interposability: Applications need to be able to coordinate with other applications that use event types unknown at the time an application was created or adapted for use in the interactive workspace. Since events are public and indirectly sent between applications, an intermediary can translate an event from a source into one or more events of different types which will cause the appropriate action in a receiver or receivers [3]. Note this can be done with message passing or RMI/RPC, as Jini does, but requires some method of ensuring applications will connect to the intermediary.

Snooping: The tuplespace model allows one component to snoop on events being sent among other components without impinging on their behavior. Information in that event can then be used to affect the local behavior of the snooping application. This is highly impractical with any client/server scheme such as RMI/RPC.

3.5 Model Summary

Figure 3 presents an overview of how the components in the system model interact. From the developer standpoint they develop their application as a set of one or more stand alone application components. Given components coordinate with the outside world by submitting tuples into the tuplespace system, and querying for tuples that match some capability of the component. Components that generate and consume similar tuples will therefore automatically coordinate among themselves, regardless of where they are in the workspace. By using the interface service system to provide a set of capabilities, application components can be controlled from any device for which the interface management system is able to create an interface. Finally, by submitting and retrieving attribute named data to the workspace wide data system application components can exchange information with any other component in the workspace for which there is a valid set of data transformers available.

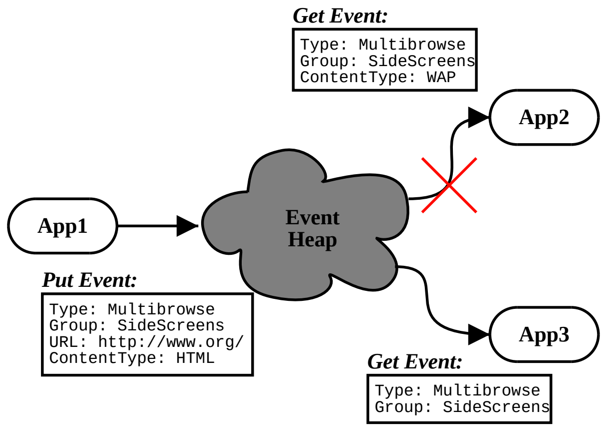

Figure 4: Basic Event Heap Interaction. App1 submits a ’Multibrowse’ event instructing group ’SideScreens’ to display URL ’http://www.org/’, content type ’HTML’. App2 is waiting for Multibrowse events addressed to SideScreens, Group ’SideScreens’, but content type ’WAP’, so it does not receive the event. App3, which can display all content addressable by a URL, does not specify a value for ContentType, so it will receive the event and display the URL.

4 Prototype System

In order to validate the system model we proposed in the previous section, we constructed a working prototype and have been using the various components in the iRoom for almost a year and a half. This section presents some implementation details for the prototype system.

4.1 Event Heap: Communication Infrastructure

The Event Heap is based on TSpaces, a Java based tuplespace implementation from IBM [30], which is in turn based on Linda [17]. An example usage is shown in figure 4.1. is given below:

Our system differs in a few key ways from a standard tuplespace model. First, multiple identical queries will yield events in FIFO order per event sender, though there is no ordering across senders. Second, our matching rules relax the constraint on field order and event size, allowing events to be submitted with extra fields unknown to some receivers without requiring the receiver applications to be rewritten; this enables receiver evolution. Third, events have a Time-To-Live field and are garbage collected after they expire; this solves the resource management issue associated with posting events that no receiver is interested in.

Although TSpaces uses a non-replicated central server for the actual tuplespace medium, we provide some robustness by effecting a fast automatic restart of the server and automatic reconnect for all Event Heap clients in case of server failure. As we will explain, many of the state-maintenance design decisions in the Data Heap and ICrafter are synergistic with making the EH restartable in this manner, making the iRoom as a whole modularly restartable [12]. The Event Heap consists of a 15KB JAR file, and we also provide interfaces in C/C++ via JNI, Python, and a Java servlet interface that converts properly-formed fat URL's into event posts and queries. This latter ability allows the creation of web pages that manipulate any of the running applications in the iRoom.

4.2 A1: Data Movement Facility

The Data Heap, the data movement facility we built as part of iROS, provides type-independent and location-independent storage of large and semi-permanent data in an interactive workspace. We use a data transformation system [**?**] to provide type-independence. This system uses a set of dynamically composable data transformers to convert data between arbitrary formats.

Since we cannot assume that all devices can access data from any one particular data source due to heterogeneity, we separate the indexing and querying of metadata from the storage of the data itself, allowing us to store data on a variety of existing data stores. Metadata-based indexing and searching of information provides location-independent naming of information in iROS, similar to the Presto system [9]. The Data Heap mediates protocol mismatches between applications and data sources by providing the capability to either dynamically download a Java protocol handler, or to have the Data Heap copy the data to a supported data source (such as a standard WebDAV server). Together, this metadata-based naming and protocol mediation provides location-independence in the Data Heap.

Metadata is stored and queried using a fast, in-memory XML database we developed, called CMX (Context Memory). Control communication (storage and querying of metadata) is handled over the Event Heap, while data communication uses the native protocols of the storage servers (we currently support WebDAV and HTTP).

Issues we have not yet addressed include consistency guarantees of data being accessed on multiple devices, and the possibility of reverse transformation to save data edited while in a transformed format.

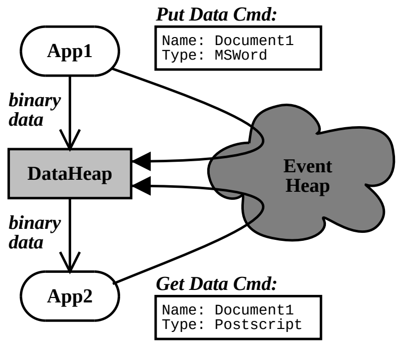

Figure 5: Basic Data Heap Interaction

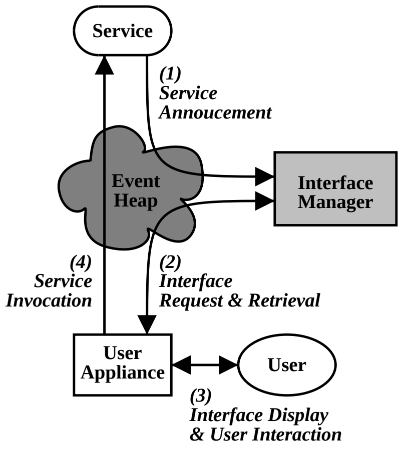

Figure 6: Basic ICrafter Interaction

4.3 A2: Interfaces Movement Facility

ICrafter, our interface management architecture, is compatible with the Interface Management System (IMS). Consistent with our overall design philosophy and goals, ICrafter architecture is entirely based on the tuplespace model of the Event Heap. Compared to similar previous work [20, 2, 13, 30], this offers the ability to snoop on and interpose between calls to ICrafter services.

Figure 6 illustrates how the ICrafter framework works. Services beacon their descriptions to the Event Heap (step 1 in figure 4.3). Service Descriptions describe the services' programmatic interfaces in an XML-based service description language. Unlike [20, 13], our service descriptions do not include URLs or physical addresses of the services or UIs, because this would require remapping whenever service/UI locations change. Due to the clean separation of service programmatic interfaces from user interfaces, the service descriptions can be generated automatically from the code for service objects written in Java (using Java reflection). This avoids the problem of maintaining consistency between the descriptions and the services as the services are updated. Requests for UIs from appliances contain the appliance descriptions, which list the UI languages supported by the appliance, and optional (name, value) pairs describing other attributes such as physical dimensions, etc.

When a request for an interface arrives from an appliance (step 2), the interface manager chooses a custom UI generator if one is available for the service and appliance platform, or a generic service-independent UI generator otherwise. The generators can access the CMX database for the local workspace context information (such as physical locations). Custom generators have been written for several iRoom services in three markup languages: HTML, VoiceXML, and SUIML (a Java Swing UI markup language developed by the HCI subgroup of our project). When a user performs an action on the user interface (step 3), the service is invoked by placing an event into the Event Heap (step 4).

4.4 A3: Dynamic Application Coordination

As described in section 2, user-directed application coordination refers to the ability of application developers to create novel applications from collections of existing Event Heap-enabled applications with no programming. Four key properties of the Event Heap: self-describing events, anonymous communication, snooping, and interposability form the basis for user directed application coordination as explained in section 3.4. Note that some of these properties (especially, self-describing nature of events and anonymous communication model) are well-known in systems literature. Our contribution here lies in identifying how these properties have proved useful in a prototype ubiquitous computing (specifically, interactive workspaces) environment and proposing how generic mechanisms can be built using them that allow users to create novel applications with no programming. By doing so, we make a strong case for adopting an underlying communication infrastructure that provides these properties in workspace-like ubiquitous computing environments.

Currently most user coordination in iRoom is ad hoc and is primarily done using two simple mechanisms:

- Debug mechanisms built into our infrastructure let users easily inspect the events used by existing applications.
- A simple web form allows users to contruct fat URLs that can be embedded in web pages and cause submission of one or more arbitrary events when clicked.

Using these mechanisms, users can easily create web pages containing links which can activate and coordinate the behaviour of several applications. For example, a link can be created on a web page which, when clicked, can cause viewers to be opened on many displays simultaneously displaying related views. Which views are shown and how they are laid out across the displays can be controlled by the user using the above mechanisms with no programming.

As another example, researchers in another subgroup designed a wireless button which can be configured with a web interface to submit a desired event. We wanted to use the wireless button to advance the slides in our SmartPowerPoint application (which will be described later). This could be achieved trivially (with no programming) by configuring the button to submit an event that caused the "advanceSlide" method to be invoked on the SmartPowerPoint service.

5 Application Examples

5.1 Smart Presenter

To showcase the capabilities of the iRoom and our software infrastructure, we have designed and implemented SmartPresenter, a multi-display, multi-object presentation program for interactive workspaces. While traditional presentation programs coordinate the display of slides across time, SmartPresenter coordinates the display of information across both time and display surfaces.

For instance, in the iRoom, with three large displays on one wall, for some specific point in their presentation, the presenter may configure the system to display an outline of the talk on the left-most display, the main content slide on the middle display, and a detail of a dataset on the right-most display. In addition, audience members may follow the presentation from a laptop, either tracking the current presentation or browsing content that is currently not being displayed. Tracking the current presentation is done by snooping on the main control events for SmartPresenter.

The core of the SmartPresenter application is a small component that reads a presentation script specifying what actions should be taken at what point in the presentation. The most common action is displaying a particular data object on a named display (such as slide #4 of a PowerPoint presentation, or a digital photograph). In addition, the script may also specify programs to run on particular displays, or even command lights to turn on and off.

SmartPresenter uses the Event Heap to coordinate the behavior of all of the display surfaces, the Data Heap to store slide information and transform slides for display on various devices (large screens, PDAs, etc.), and is controlled using ICrafter-generated interfaces. SmartPresenter viewers are implemented as Event Heap-aware display services, wrapping legacy viewer applications (such as Microsoft Powerpoint or Internet Explorer).

Together with the decision to reuse existing software for the display of data, our infrastructure greatly simplified the development of the SmartPresenter. The Event Heap handles the communication among the distributed components, the Data Heap manages the transfer of data to each display and the formatting of data for the various displays, and ICrafter creates simple, usable interfaces for various devices. All that was left to build was the application logic for the SmartPresenter.

5.2 CivilE Suite

An early adopter of iRoom technology has been the [ANONYMOUS] (CivilE), a group of civil engineering researchers investigating how to better automate the process of planning and management for the construction of large civil engineering projects. Discussions with CivilE inspired the scenario with which we opened the paper and have resulted in the CivilE application suite, which demonstrates how a collection of application components can be created that permit the user to determine which components are coordinating at any given time. CivilE designed a set of viewers for their data that could be run on the various displays in the room:

- A construction site map that allows the selection of various view points in the construction site and then emits an appropriate view change event.
- A “4D” viewer that shows a 3D model of projected state of the construction site for any date during construction. It responds to view change events, object and zone selection events (e.g., building 3), and display time events.
- Another map viewer that highlights zones based on zone selection events.
- An applet based web viewer that displays tables of construction site information and emits zone and date selection events as table information is selected and listens for the same events to select information in the table.

All of the applications use the Event Heap for their communication, and in fact began life as separate, non-coupled applications. Since they use common event types, the various components of the suite retain their ability to coordinate while still being able to be brought up on any screen in the room. Since the components are loosely coupled, if no event source is running, or there is no event sink, it does not affect any of the application components currently in use. Much of the CivilE application was essentially plain HTML using properly constructed fat URL’s; only the custom 4D viewer had to be coded specifically to communicate with the Event Heap.

5.3 Other Applications

The SmartPresenter application and the CivilE Suite between them give concrete examples of how actual applications use the three Application Developer Facilities (A1-A3) introduced in section 2.1. There are many other applications that have been written using the infrastructure, and we list a few of them here:

Image management system: Images can be placed into a workspace wide repository from laptops, permanent iRoom machines, or directly from scanners. They can be displayed anywhere in the iRoom.

Room controller: Actually an ICrafter interface, it provides a geometric view of the room, the ability to move data to any display in the room, and control over lights, projectors, and routing laptop video to screens.

Radiology Visualization: Allows users to view 2D slices of a 3D volumetric data set. Viewers run on independent machines.

Storyboarding Application: Lets film storyboard artists sketch pictures, scan them at a scanning station, manipulate them using a image sorter application on another screen, and move them to Adobe Premiere for creating a movie on two other adjacent displays.

Wireless buttons: We built some simple wireless buttons that communicate with a base station connected to a computer in the iRoom. In order to connect this wireless button infrastructure to iROS, we added event posting capabilities to these buttons. This integration was achieved with less than 20 lines of code and immediately opened up a number of new functionalities for the existing applications in the iRoom. These buttons are now used for controlling lights and projectors and for controlling applications such as the SmartPresenter described in section 5.1.

Although we have presented these as “applications”, since the components were constructed using our infrastructure, they can flexibly interoperate with one another. For example, the same image viewer is used in the image management system, the radiology visualization and the storyboarding application—there are just different sources of the request for the viewer to display the information. This is important since we imagine providing users with a set of tools (component applications), they can flexibly arrange to create unique applications on the fly.

6 Evaluation

Building ad hoc software to provide the functionalities we have described is not especially challenging in and of itself; what makes it challenging are the additional system goals of high robustness, portability/reusability, and extensibility. The evidence we present in this section suggests that we have made encouraging progress in meeting these goals. We also quantify the performance of key parts of iROS and identify design or implementation changes that would improve it.

6.1 Robustness

iROS was designed and built to avoid cascading failures, and to be robust to individual failures. We repeatedly employ three main techniques to achieve this:

Loose Coupling: Since software entities communicate through the Event Heap but are otherwise autonomous, failures do not generally cascade. In the SmartPresenter application, for example, the failure of one or more displays does not affect the presentation of data on other displays. Applications that attempt to do blocking RPC over the Event Heap eventually time out if there is no response, with the stale events themselves eventually being garbage collected. This style of communication encourages programmers to think in terms of autonomous message-passing entities rather than in terms of clients and servers whose states are tightly coupled through procedure calls.

Announce/listen rather than Register/Deregister: For example, in ICrafter, not only do we beacon service advertisements rather than requiring registration and deregistration, but we also make the beacons themselves carry the service descriptions. This solves two potential robustness problems: first, if a service dies unexpectedly, other components will soon notice it has stopped beaconing, so there is no issue of resource reclamation as there would be if a service failed to deregister gracefully. Second, there is no concern that a stored persistent version of a service description might become inconsistent with what the service can actually do.

Modular Restartability: The above properties, combined with automatic restart and reconnection of the Event Heap, allow iROS as a whole to gracefully survive transient failures of almost any subset of its components, and recover from them when the failed components are restarted. We have found this to be critical in practice for keeping the iRoom running; in steady state, with no applications running and not counting basic Windows and Linux services, our iRoom is running about 30 process and nearly 200 threads across 8 PC's.

6.2 Extensibility

Our infrastructure is designed to be extensible in two ways, by adding devices and by adding services:

Devices: There are three aspects of integrating a new device into iROS:

1. The device and its applications must be able to use the Event Heap. This can be done by writing applications linked against Event Heap libraries in one of various languages, by running its applications a generic runtime wrapper that supports the Event Heap (as ICrafter does for generating generic interfaces for Win32 applications), or by browsing appropriately-constructed Web pages and forms on the device's browser.

2. The device can optionally add any type transformers necessary to convert from at least one common data type to a format understood by the device. For example, the only iROS-visible API for displaying information on the Hires Mural is a transformer that converts any JPEG into the appropriate OpenGL sequences for rendering bitmaps.

3. A device that wants to display human interfaces can optionally add a custom interface generator for itself to ICrafter. It can also use an existing generic one, such as the HTML interface generator.

Services: In order for a new service to become "ICrafter-enabled" (remotely controllable through a dynamically-generated UI), the service's Java class (or wrapper) must be registered with ICrafter, or the service must be of a type whose methods can be automatically introspected by ICrafter (e.g. using reflection). Service descriptions can then be automatically generated and optionally hand tuned for higher quality. Similarly, the user interface can be handcrafted or a generic user interface can be automatically generated (e.g., HTML).

To date, we have successfully extended the iRoom with new, unplanned, devices, such as the wireless button described in section 5.3. We have built a large number of services, from simple light controllers to more complicated meta-applications like the SmartPresenter and CivilE construction management demo, with little new code. The ability to rapidly add a new device or service for prototyping, and hand-tune the integration later, has been a valuable property for a research testbed in ubiquitous computing.

6.3 Portability

We have successfully transferred iROS to partner sites that are using it as "customers" rather than researchers, i.e. their interest is in deploying their own domain-specific applications rather than contributing to ubiquitous computing generally. Researchers at the ANONYMOUS use the iRoom to demonstrate how construction project teams could use a multi-device and multi-display environment to display and interact with project information. Their initial software consisted of a number of independent data viewing applications; the iRoom version of their demo allows each viewer to run on its own display, and coordinates their actions using the Event Heap. CivilE is also deploying an "iRoom to go" using multiple laptops and a subset of iROS.

We have also deployed a subset of iROS to the ANONYMOUS (AAA). Their prototype interactive workspace includes two Smartboards and a laptop. Once we identified which files were needed, we were able to copy and use them without modification (with the exception of some web server configuration files). AAA now plans to start user-testing and to setup technology-assisted-learning applications and scenarios that use this infrastructure. We have also successfully run the Java-based Event Heap and related servers on Windows 2000 (in the iRoom they run on Linux).

Though preliminary, our experience to date indicates that we are on track to satisfying our goal of portability of our meta-operating-system both to different iRoom installations and across traditional operating system platforms.

6.4 Relevant Measured Performance

In this subsection, we report the performance observations for our system. Note that since our prototype implementation is primarily an initial proof-of-concept implementation, we have not yet worked on performance optimizations. Our primary goal was to build a system that was fast enough for human perception ranges.

As mentioned earlier, the Event Heap is not intended for low latency fine-grained GUI-like events (we have another system for this purpose). Instead, it is primarily intended for high-level control events.

We measured the scaling performance of the Event Heap by observing the latency of the Event Heap under varying loads. The load on the Event Heap was controlled by running a number of client processes on a cluster that accessed the Event Heap in various ways (the clients performed submission, retrieval and removal of events). The TSpace server was run on a 4-cpu linux server, and configured to use 256MB of physical memory. With no load on the TSpace server, we measured a latency of 30ms. However, as the load increased, we noticed a wide variation in the latency depending on several factors such as the duration for which the TSpace server was running, the number of tuples in the server, configuration settings for the TSpace server, etc. For example, with 200 clients, latency varied from 100ms to 2 seconds.

Tspaces is a feature-rich research product. (It provides transaction support and also much more powerful query features than those used by the Event Heap.) We believe that a ground-up implementation of the Event Heap (or re-implementing it on a simpler and more mature commercial blackboard system) would result in much better performance. Currently, because the iRoom rarely has more than 10-20 active Event Heap clients simultaneously, we have found the performance to be acceptable.

We had similar experiences with the Data Heap System. While our performance numbers varied significantly based on the format and size of the data, CMX performed satisfactorily for a few thousand entries at a time.

To evaluate the performance of ICrafter, we measured the end-to-end interface generation time for a typical web interface. Ignoring the browser rendering time, The time taken for this can be broken down into HTTP latency, Event Heap latency, and the actual interface generation time. The observed performance was as follows:

Total time:	555ms
HTTP latency:	44ms
EH latency:	70ms
Interface generation:	441ms

Interface generation time is fairly high because the critical path includes several transitions between java and python interpreters. Thus, the performance is heavily dependent on the python interpreter and the overhead of the transitions. While better performance is definitely desirable, we have found the current numbers to be acceptable.

However, our most interesting results came about from our study of the applications written for the iRoom. Our observations proved that iROS enabled application developers to write iRoom aware applications using very few lines of extra code. Table 1 lists our observations. In most cases, the amount of new code required to “enable” a component to use the Event Heap, Data Heap or ICrafter was on the order of tens of lines or a few hundred statements. Most applications required zero code to become ICrafter-enabled, since their interfaces can be generated completely automatically. The work-to-benefit ratio of enabling new applications has been uniformly very favorable.

6.5 Summary

We designed iROS to be both a useful testbed for ubiquitous computing research and the basis of “production” ubiquitous computing environments. The ability to quickly deploy and integrate new devices and services without having to hand-tune interfaces until later has been very valuable in evolving the iRoom. Similarly, the defensive loosely-coupled design relieves us of having to worry that introduction of a new (buggy) device or service will destabilize the environment. The cost of this design is the dependence on a centralized communication medium (the Event Heap) and the potential performance pitfalls associated with it, but we are confident that relative to our naive implementation, this hurdle can be overcome and the benefits of the simplicity and robustness of the resulting programming model readily reaped.

Application	Total Lines	No. of ';'	Event Heap code	Data Heap code	ICrafter code
SmartPresenter	993	341	40	10	0
Button	300	232	8	n/a	0
Butler	298	107	10	n/a	7
Light Controller	113	41	0	n/a	0
Projector Controller	450	148	25	n/a	25

Table 1: Code analysis of applications written using iROS

7 Discussion and Synthesis

We believe three aspects of our design are fundamental to this style of ubiquitous computing, independent of iROS:

1. Data and interface movement abstractions with infrastructure support;
2. Group communication using a centralized infrastructural implementation (as opposed to, e.g., multicast and distributed state);
3. Loose coupling as the only practical way to achieve *both* robustness and extensibility.

We consider each in detail.

7.1 Data and Interface Movement Are Fundamental Abstractions

Data movement and interface movement abstractions are fundamental because they span the axes of extensibility. Data movement allows the integration of new applications; interface movement allows the integration of new devices. Although analogous techniques for moving data and interfaces exist in single-node OS's and toolkits, applying the ideas to ubiquitous computing requires generalizing them in important ways.

Traditional GUI OS's and toolkits such as Win32 and X11 [25] have long relied on a "clipboard" metaphor to enable data communication between applications not designed to interoperate. Typically, a data producer must post data in multiple formats: its native format, plus one or two "canonical" formats guaranteed to be understood by other applications. For example, a spreadsheet might post both a spreadsheet object and ASCII text. This approach works well on single-node OS's since applications written to that OS have already evolved a set of datatype conventions. The Data Heap generalizes the clipboard concept in two important ways. First, the transformation facilities are automatic and extensible: it automatically computes multi-step transformations if necessary, freeing the data producer from anticipating the possible needs of consumers. This is important because of the much greater degree of heterogeneity in a ubiquitous computing environment: the consumer of the data might be a quite alien application on a very different platform. Second, we move both the storage and the transformation logic into the infrastructure, rather than making them part of any client, which serves to keep the client simple.

A similar argument can be made for ICrafter. Applications written for a particular GUI may be coded to keep the UI separate from the application logic, but the UI itself is usually expressed directly in the facilities provided by the toolkit (e.g. buttons and windows in Win32), or at best in a slightly more abstract language that can be mapped to a variety of implementations, as is the case in Geoworks and more recently XSL/XSLT. ICrafter generalizes this approach by moving interface generation and service discovery into the infrastructure, freeing the client from every aspect of remote control except requesting and instantiating the interface.

7.2 Centralized Group Communication Enables Incremental Integration

We have had some success with scenarios of "incremental integration": given an ensemble of n existing software components that already work together, add a new component that can either control or influence the behavior of the ensemble, or react to behaviors of other ensemble members. Both capabilities require some form of multicast-like communication—the former to address control messages to ensemble members, the latter to "snoop" messages among ensemble members and use them as external stimuli. We could have used a distributed-state solution such as Scalable Reliable Multicast [11], but putting a centralized tuplespace in the infrastructure offers two important advantages. First, it provides a degree of failure tolerance and flexibility: since events can persist after being posted, it decouples communication in time and allows a new ensemble member to rapidly "catch up". Second, it keeps clients simple: only the trivial Event Heap logic, and not distributed state management, need to be incorporated into client-side libraries. This is an important benefit given the collection of diverse platforms we wish to support. Further, our problem size (hundreds of devices) does not require the scalability benefits of schemes such as SRM.

As currently implemented, the Event Heap is a single point of failure. We note, though, that there is nothing preventing us from adopting a more robust implementation—in fact, it was recommended [27] that we try a commercial database, whose performance and capacity are more than adequate given our latency requirements and problem size. Furthermore, the Event Heap is automatically reconnected after a transient failure, and we do not rely on event persistence except as an optimization (e.g. to pick up service advertisements without waiting for all services to beacon again). In practice, we have found that living with the communication substrate as a single point of failure has not been a problem, even with the iRoom in daily continuous use and with component failures occurring a few times per day.

7.3 Loose Coupling Connects Sophisticated Building Blocks

That loosely-coupled message passing preserves existing fault isolation boundaries is particularly important in a dynamic environment such as an interactive workspace: we can quickly experiment with new devices and software without concern that we will destabilize the environment. We believe the *combination* of robustness and interoperability we have been able to realize would not have been practical any other way. In fact, the predecessor to iROS had these problems: it was fast, but not extensible (adding new devices or services was ad hoc) and not robust (the components themselves were buggy, and the system was not resilient to component failures, frequently wedging or requiring manual intervention). We cannot prove that loose coupling has made the only difference, but given our experience with iROS, we are certainly willing to pay the performance cost of indirect communication in exchange for the other benefits.

Loose coupling has also allowed us to build sophisticated behaviors by starting with extremely simple but powerful existing mechanisms, leveraging the full power of each node's OS and applications. For example, we use URL's as global names within a single work session; we leverage Web browsers' ability to dispatch content to the correct viewers on their respective platforms; we use HTML and Java Swing as simple UI prototyping environments and WebDAV as a simple networked storage mechanism with `/tmp` semantics; and we rely directly on PDA transformation proxies [14] to make these features available directly on handhelds with no additional programming. Such mechanisms are well-understood, enjoy widespread support in new and existing devices, and are easy to implement when unsupported. The "raw" results of using these mechanisms without new custom code are usually not the most elegant that can be achieved, but for each of the above example mechanisms, we provide a corresponding way to handcraft a solution at any desired level of detail that successfully interoperates with automatically-generated solutions. We have already expended much effort in system integration; as the number of components to connect (and keep running) increases, it has been useful to leverage these efforts by simply plugging in new devices and software that use tried-and-true interfaces. It is difficult to capture the ease with which we have been able to confidently and rapidly integrate new devices and services into the always-evolving iRoom.

8 Future Work

Application coordination. The user-directed application coordination aspect of our infrastructure is the least mature. Currently we lack a formal model for describing application coordination, and there is no notion of what constitutes "correct" behavior for the rest of the application if one component fails. (We have focused on protecting the rest of the system from faults rather than monitoring the correctness of individual applications.) We began investigating a more formal and systematic approach to dynamic control flow based on state machine representations [3], but have not had the resources to pursue this avenue of research further.

Scalability. So far, we have only focused on developing infrastructure software to enable a single-room installation. The fundamental limits on scalability imposed by the physical size of a single meeting room (number of occupants, number of physical devices, etc.) have allowed us to trade wide-area scalability for simplicity, robustness, and ease of programming in the design of the software infrastructure. We are beginning to investigate how collaboration might occur across multiple interactive workspaces; we anticipate having to create separate mechanisms to propagate relevant events between distinct Event Heaps in each interactive workspace. As with our original approach, we would like to observe how people wish to collaborate across interactive workspaces before identifying what the right abstractions are. We expect to have this opportunity as the two new workspaces come online at our partner sites.

Security. This is an unaddressed problem, in part because we lack a social model to indicate what security mechanisms would be appropriate in collaborative settings: when a user moves information from a personal device to a large screen, it becomes publicly readable; when a user has the ability to control public infrastructure remotely from a personal device, there is a tradeoff between user convenience and authentication (as is typical in security systems). To complicate matters, our legacy-OS building blocks, such as Unix and Windows, assume (differing) single-user-at-a-time models for controlling the console display and allocating privileges. The current situation is crude: the room is firewalled from the outside and reasonably physically secure, but once inside the room there is no other authentication or access control (except as required by specific applications, e.g. if an iRoom user attempts to open files from a remote fileserver). A fictitious user with minimal privileges is permanently logged in at all the machines that control public infrastructure.

9 Related Work

Previous systems have addressed individual mechanisms such as low-level communication mechanisms, discovery (Jini [2] and UPNP [13]), naming (INS [1]), and adaptation ([23]. Our focus has been on the combination of the facilities needed by the application developer. Consequently, we only evaluate here other systems that attempt to comprehensively address the software infrastructure required for specific workspace-like ubiquitous computing environments. Past work that is relevant to only one of the communication infrastructure, data management, or user interface management subsystems has already been addressed in sections 3 and 4.

Most other workspace-like projects (in other literature also called smart rooms, intelligent rooms, etc.) have not specifically focused, as we have, on building an operating environment possessed of all four qualities: reusable, robust, portable and extensible. The Intelligent Room at the MIT AI Laboratory [6] and Microsoft Research EasyLiving [5] both use a combination of sophisticated sensor fusion and AI techniques to enable the environment to deduce the user's needs from contextual and other cues. Adding new functionality to EasyLiving is currently *ad hoc*. "Smartness" was not one of our goals: we focused instead on providing the infrastructure for application programmers to simplify writing applications with the behaviors they desire.

Our observations of user behavior are similar to those made by i-LAND [28]. This mature project, whose physical space is very similar to our own, has identified design goals closely related to user-directed application coordination and movement of information. The primary difference between i-LAND and the iRoom is the approach to software infrastructure: they have written tightly-coupled custom applications based on a SmallTalk-specific CSCW framework called COAST [26], whereas we focus on loosely coupled generic mechanisms that enable integration of legacy/COTS hardware and software. In addition, they provide no explicit support for automatic transformation during data movement, interface adaptation, or application coordination with no programming, all of which we believe should be first class goals for any system infrastructure for an interactive workspace.

The Portolano project at the University of Washington is exploring some similar issues as iROS. Their current work on an instrumented and enhanced biology lab workbench [10] is similar in spirit to our iRoom. Their current efforts focus more on the facilities a lower-level programming model such as One.world [19] should provide.

Finally, we have stolen from well-known previous work the principles of loose coupling for robustness, soft state, and announce/listen protocols [8, 16]; the case for infrastructure-centric approach to adapting to client and network heterogeneity [15]; and the case for a "systems of systems" approach [18] in which we connect legacy building blocks consisting of complete systems with OS's and applications.

10 Conclusions

We have focused on one type of ubiquitous computing, the interactive workspace. We have identified what we believe to be fundamental higher-level abstractions for room-based ubiquitous computing: moving data around, moving interfaces around, and coordinating the behavior of monolithic applications. These abstractions are fundamental because they span the space of extensibility: data movement allows the integration of new applications, user interface movement allows the integration of new devices. We have discussed our experience with iROS, a deployed meta-operating-system embodying these ideas, and our experience creating new applications and retrofitting legacy applications to work with iROS.

In the interest of keeping clients robust, we rely on indirect communication through a tuplespace, which preserves fault isolation boundaries; in the interest of keeping them simple, our abstractions are implemented as infrastructure software, with which other entities communicate using simple and platform-neutral network protocols. The combination of robustness, extensibility, and ease of programming we have been able to achieve would not have been practical without this loose coupling, and the users deploying our system in additional interactive workspaces are so far validating our experience.

Despite software and hardware advances and falling costs, ubiquitous computing has been hampered by the lack of a uniform set of appropriate higher-level abstractions that also result in a robust system. We offer our lessons with iROS and the software artifact itself with the hope that it will be a first step in accelerating progress in ubiquitous computing research.

Acknowledments have been blinded for review.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. 33(5):186–201, December 1999.

[2] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison Wesley, 1999.

[3] Authors blinded for review. Dynamic control in tuple spaces for sustainable evolution in pervasive applications. Unpublished extended abstract.

[4] Authors blinded for review. Distributed rendering for scalable displays. In *IEEE Supercomputing 2000*, 2000.

[5] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. Easyliving: Technologies for intelligent environments. In *Handheld and Ubiquitous Computing 2000 (HUC2K)*, sep 2000.

[6] M. Coen. The future of human–computer interaction or how i learned to stop worrying and love my intelligent room, 1999.

[7] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Third USENIX Symposium on Internet Technologies and Systems (USITS 01)*, San Francisco, March 2001.

[8] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei, P. Sharma, and A. Helmy. Protocol independent multicast (PIM), sparse mode protocol: Specification, March 1996. Internet Draft.

[9] Paul Dourish, W. Keith Edwards, Anthony LaMarca, and Michael Salisbury. Uniform document interactions using document properties. pages 55–64, November 1999.

[10] Larry Arnstein et al. Ubiquitous computing in the biology laboratory. *Journal of Laboratory Automation*, March 2001.

[11] Sally Floyd, Van Jacobson, C. Liu, and Steven McCanne. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. In *ACM SIGCOMM '95*, pages 342–356, Boston, MA, Aug 1995.

[12] (Blinded for review). Modular restartability. Submitted for publication.

[13] UPnP Forum. Universal plug and play. Available at `http://www.upnp.org`.

[14] Armando Fox, Ian Goldberg, Steven D. Gribble, Anthony Polito, and David C. Lee. Experience with Top Gun Wingman: A proxy-based graphical web browser for the Palm Pilot PDA. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, Lake District, UK, September 15–18 1998.

[15] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Adapting to network and client variation using active proxies: Lessons and perspectives. *IEEE Personal Communications (invited submission)*, Aug 1998. Special issue on adapting to network and client variability.

[16] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, St.-Malo, France, October 1997.

[17] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, pages 80–112, January 1985.

[18] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix: A Global Layer Unix for a network of workstations. *Software Practice and Experience*, 28(9):929–961, 1998.

[19] Robert Grimm. Personal communication, February 2001.

[20] Todd Hodes and Randy Katz. A document-based framework for internet application control. pages 59–70, October 1999.

[21] Emre Kiciman and Armando Fox. Using dynamic mediation to integrate cots entities in a ubiquitous computing environment. In *Handheld and Ubiquitous Computing (HUC 2000), First International Symposium*, September 2000.

[22] A. Gefflaut M. Roman, J. Beck. A device-independent representation for services. December 2000.

[23] Brian Noble and Mahadev Satyanarayanan. Experience with adaptive mobile applications in odyssey. 4(4):245–254, August 1999.

[24] John Ockerbloom. *Mediating Among Diverse Data Formats*. PhD thesis, Carnegie Mellon University, January 1999.

[25] Bob Scheifler and Jim Gettys. *The X Window System*. MIT Press, 1987.

[26] Christian Schuckmann, Lutz Kirchner, Jan Schummer, and Jorg Haake. Designing object-oriented synchronous groupware with coast. November 1996.

[27] Steven Shafer. Personal communication. 2001.

[28] Norbert Streitz, Jorg Geibler, and Torsten Holmer. Cooperative buildings - integrating information, organization, and architecture. pages 4–21, February 1998.

[29] Mark Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–100, September 1991.

[30] Peter Wyckoff, Stephen McLaughry, Tobin Lehman, and Daniel Ford. Tspaces. *IBM Systems Journal*, 37(3), August 1998. Available at `http://www.almaden.ibm.com/cs/TSpaces`.