Designing for Maintainability, Failure Resilience, and Evolvability in Ubiquitous Computing Software

Shankar Ponnekanti, Brad Johanson, Emre Kıcıman, and Armando Fox
Computer Science Department, Stanford University
{pshankar,bjohanso,emrek,fox}cs.stanford.edu

Abstract

The design constraints in ubiquitous computing (ubicomp) differ from those traditionally emphasized by the systems community: evolvability, long-term maintainability, and robustness to transient failures are essential, while scalability and performance are lesser concerns, due to the nature of ubicomp itself and the performance of today's commodity equipment. We show how these observations are reflected in the design of iROS, a ubicomp software framework in production use. In particular, we show that a centralized architecture directly enables the ubicomp programming abstractions needed while providing the best solution for evolvability and maintainability/deployability, and that we can achieve the required robustness through a fast-recovery strategy, which allows a simple centralized implementation of the architecture. Throughout, we achieve performance, scalability, and recovery behavior sufficient for typical operation.

1 Introduction

Ubicomp is an important emerging field, one of whose major challenges is system software [17]. We argue that in ubiquitous computing environments such as the iRoom (our research testbed), the design goals are different from those the systems community has traditionally considered. In particular:

- Although maintainability, evolvability, and resilience to failure and other dynamic conditions are always desirable, in a ubicomp environment they are *essential* because of the nature of the environment itself; and
- although high performance and scalability are always desirable, current hardware and software makes the performance of a centralized ubicomp architecture adequate for human-scale latencies, and the naturally-limited scale of ubicomp makes scalability a lesser concern.

These observations lead us to a simple (and therefore deployable and low-maintenance) system architecture with *sufficient* performance and scalability that nicely supports important programming abstractions required for room-scale ubicomp.

This paper describes why ubicomp environments lead to potentially different design decisions, and how they were made in iROS (the Interactive Room Operating System), the software infrastructure for a ubicomp environment that has been in production use for 2+ years.

Since ubicomp is a relatively immature field, comparisons of specific mechanisms within ubicomp programming frameworks are hard to make. Unlike the operating systems community, ubicomp practitioners have not yet settled on a small set of primitives and the corresponding implementation tradeoffs and micro- and macro-benchmarks. We propose a set of ubicomp "primitives" we have found useful as the basis of our evaluation. We do not claim our set of primitives is authoritative or exhaustive, but they are based on more than 2 years' experience with the iRoom and about 1 years' experience with its more primitive predecessor, and they have enabled the development of demonstrably useful ubicomp applications.

Our goal is to demonstrate that a simple, centralized architecture directly enables the evolvability and ease-of-maintainability goals of ubicomp; that a rapid-recovery scheme (rather than a distributed implementation or other use of functional redundancy) suffices to provide the needed robustness; and that the scalability and performance of the resulting implementation are sufficient for this application domain despite the design choice of centralization.

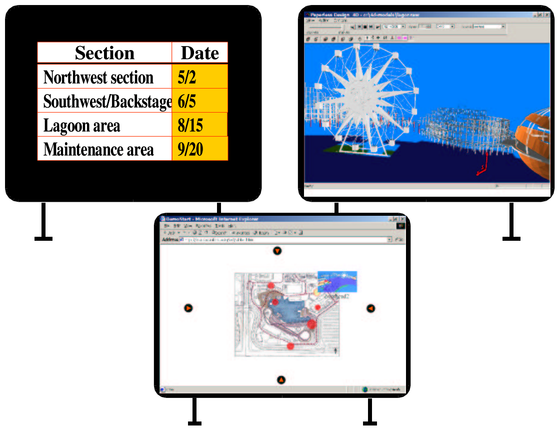
In the next section, we describe the unique constraints of ubicomp, and introduce the design decisions we made to meet these constraints and satisfy our goals. Section 3 highlights the salient features of iROS that illustrate our design decisions. (Since each of the three iROS subsystems is the subject of one or more full papers, we are forced to omit many details of their implementations here.) Section 4 describes the specific design choices and quantitatively evaluates their effectiveness in meeting the goals of evolvability, maintainability, and failure resilience, while providing sufficient performance and scalability. Section 5 discusses our positive and negative experience with a wide variety of iROS-based applications and other deployments of our software, including open problem areas. We discuss related work in section 6, focusing on other complete ubicomp frameworks and other systems that share our "design for recovery" philosophy. We summarize our design choices and contributions and conclude in section 7.

2 Ubicomp Has Different Priorities

As an example of a ubicomp environment, we focus on an *interactive workspace (IW)*: a localized technology-augmented environment where people come together for collaborative work. Our testbed, the iRoom (figure 1), features three rear projected touch-sensitive screens along one wall, a bottom projected table, and a custom 12-projector tiled display ("the Mural" [11]) driven by a workstation cluster that does distributed rendering of OpenGL. The iRoom is the second generation of such an environment that we have built and experimented with. We invited several non-CS research groups to prototype scenarios and applications in this environment, to better understand how it would be used and what programming facilities we should provide. Consider the following scenario in the iRoom, representative of what we observed and illustrates specific behaviors requiring programming support.

A group of construction management engineers and contractors is holding a meeting to plan a construction project. Participants enter the room and turn on their laptops containing 802.11b wireless cards, but do not perform any particular "login" procedure. The group leader turns on room lighting and touchscreens using a Java-applet-based UI on her laptop. A second group member uses a drag-and-drop display control interface on her laptop to display an aerial view of the construction site on the table display. Similarly, a 3D wireframe model of the construction site is displayed on one of the large touch screens, and a project schedule spreadsheet

Figure 1: A meeting in the the iRoom



Section	Date
Northwest section	5/2
Southwest/Backstage	6/5
Lagoon area	8/15
Maintenance area	9/20

Figure 2: A construction management application ensemble. The screenshots are stylized for clarity, but the application is real.

on another. (See figure 2.) One of the members brings up a view of the model on her PDA, but since the model is too complicated, a simplified non-interactive image depicting the view is shown on the PDA.

Each largescreen viewer can function as a stand-alone application, but when a user selects or makes changes in one view, the other views update themselves to show information relevant to the selected view. For example, selecting a cell on the spreadsheet schedule causes the wireframe view to render the parts of the project that will be completed by that date; selecting a point on the aerial view rotates the wireframe model to show the corresponding vantage point; selecting a structure on the wireframe model highlights the cells in the schedule spreadsheet that are milestones in completing that structure. All such selection can be done either by a user interacting directly with a touchscreen or by any of the laptop users "remote controlling" any touchscreen using a single mouse, in effect giving the illusion that all the touchscreens and a given user's laptop are combined into a single large logical display surface. Further, any user can "pull" one of the viewers to her own laptop screen and interact with it locally to get the same effects. During the meeting, one user must leave unexpectedly; she hastily shuts down her laptop (without any "logout" procedure) while data viewers are still running on it.

2.1 The Volatility and Boundary Principles

The above scenario highlights two properties that distinguish ubicomp as a distinct subarea of distributed systems. The two properties have been previously described [17] as follows:

1. **The Volatility Principle (VP):** The set of participating users, hardware and software components in a ubicomp environment is highly dynamic and cannot be predicted in advance. The sudden departure or arrival of a service, device, or user should be considered normal operation, not an exceptional condition or a failure requiring special handling.

2. **The Boundary Principle (BP):** Ubiquitous computing environments are bounded in extent; the boundaries may be physical (the walls of a room) or cultural/administrative (there may be nonlocal resources that are acceptable to use, and vice-versa). These boundaries should be visible to applications running in the environment, to make commands such as "turn on all the lights" or "display this data on the center screen" meaningful.

These two principles lead to three specific consequences of VP and BP that must be addressed by a ubicomp software framework:

VP1: Failure Resilience. Volatility on smaller time scales requires us to deal with dynamism (e.g. people or devices entering/leaving spaces without signoff) and partial failures as common cases. Transient failures in parts of the system should not cause cascading failures, and recovery from transient failures should not require unavailability or recovery of the whole system. In particular, human users should not perceive an obtrusive level of unavailability in response to such cases.

VP2: Evolvability/Deployability Volatility on larger time scales implies that incremental evolution/accretion and therefore extreme heterogeneity will be the norm in these environments [5]. Furthermore, due to the existence of useful large building blocks and technologies such as the Web, desktop/productivity applications, etc., ubicomp software must make it easy to create new applications and behaviors from existing pieces. It is not hard to see that evolvability and legacy support are two sides of the same coin; supporting them requires the ability to integrate and leverage entire systems (OS plus applications), some of which may not have been designed for integration. "Java everywhere" and similar approaches do not suffice, because they attempt to define heterogeneity out of existence and assume that non-conforming applications will be rewritten. Techniques for dealing with heterogeneity have been well explored in the mobile computing and Web communities, but leveraging them requires that the framework must minimize the work required to integrate each new platform. Finally, since heterogeneity implies instability, VP2 reinforces VP1.

BP1: Application-level enforcement of boundaries. The end-to-end argument [24] suggests that the boundaries referred to by BP cannot be enforced solely at network level, especially when the physical or administrative boundary does not correspond directly to any technological boundary (time-to-live of IP multicast, reach of a wireless network). Mechanisms visible to applications must make the boundaries of the environment explicit; e.g. if a facility analogous to broadcast is provided, application-level mechanisms must be used to correctly limit its scope to the current workspace.

From a deployability standpoint, a system is more likely to be widely adopted if its architecture is easily understood, and if its robustness mechanisms are simple enough to give high confidence in the invariants they embody. We were therefore motivated to find the simplest architecture we could that satisfied all the above constraints.

2.2 Design Decisions Resulting from VP and BP

Software infrastructure for ubicomp must address the above constraints, in addition to providing support for programming abstractions common in ubicomp environments. The next section describes what those abstractions are and how they are supported in iROS, but first we note that the above constraints are set against two opportunities:

- Since BP inherently limits the scale of a ubicomp environment, a design may be optimized for evolvability or robustness rather than extreme scalability.

- Current hardware is so fast that performance against human-scale latencies should not be difficult to achieve, so a design may be optimized for evolvability or robustness rather than very high performance.

Of course, we must provide *sufficient* scalability and performance for typical operation. With the above in mind, we summarize the design decisions described in the rest of the paper. The common thread running through them is that a centralized architecture directly enables the programming abstractions we need, while providing the best solution for evolvability (VP2) and enforcement of application-level boundaries (BP1). We can achieve the required robustness (VP1) through a fast-recovery strategy, which allows a simple centralized implementation of the architecture. Throughout, we achieve performance, scalability, and recovery behavior sufficient for typical operation.

1. Levels of indirection and placing functionality in the infrastructure (VP1 and VP2). By placing most of the sophistication of iROS into infrastructure software (i.e. running on fixed servers in the iRoom) rather than distributing it among clients or components, we provide levels of indirection in communication, data transfer, and end-user control of services. The level of indirection in communication leads to loose coupling between components and therefore better failure isolation. Levels of indirection in our data movement subsystem and human-interface generation subsystem enable easier evolution to new data formats and UI languages/modalities. Putting the sophistication of our integration mechanisms in the infrastructure also makes new devices and new or legacy software easy to integrate. The increased latencies resulting from indirection are not significant under typical operation.

2. Logically-centralized communication model for evolvability (VP2). Entities in iROS communicative via a tuplespace-based, shared, event-driven communication model; that is, all events are potentially visible to all entities. We illustrate how features of the tuplespace model such as the shared event notification channel and the ability to do snooping and intermediation of events enable easier integration of new devices, and therefore improve evolvability. A logically-centralized communication abstraction also provides an application-level mechanism for scoping applications to the workspace: the extent of the environment is defined as the set of entities communicating via a particular instance of a tuplespace. The limited scalability implied by centralization is not significant under typical operation, due to the limited scale implied by BP.

3. Centralized implementation of communication model. This design choice addresses VP2 and BP1. A centralized (single-server) implementation of communication is much simpler to build and maintain, allows clients to be simpler by putting the communication logic in the server rather than in each client, which makes integration of diverse (possibly resource constrained) clients easier.

4. Simple restart based recovery using auto-reconnect. A centralized implementation is generally considered undesirable because it is a single point of failure. Rather than adding failover or standby redundancy to the communication implementation, we augment the single-server implementation with an auto-reconnect feature that enables clients to recover automatically when the communication server is restarted following a transient failure. The result is a measurable but tolerable user-perceived additional latency during recovery. We argue that the combination of sufficiently good recovery behavior and overall simplicity makes wide deployment more likely.

5. Use of beaconing and soft state. Components in iROS use beacons to announce their “upness” and other runtime information. Unlike approaches based on registration/deregistration of entities, with soft state beaconing, the right thing happens when a component enters or leaves the system unannounced. Because beacons are soft state, components do not run any special recovery code, making their implementations simpler and their recovery faster, and allowing partial recovery (by restarting only those components that failed) rather than requiring a recovery of the entire system when any component fails. After a failure, a user may perceive transient inconsistency between which components are advertised as available and which ones actually respond when addressed; we show that our system can scale to sufficiently high beaconing rates to achieve acceptable consistency levels under typical operation.

The result of the above design decisions is a simple set of mechanisms that simplifies operation/administration, evolvability, and deployment/long-term maintainability.

Table 2.2 summarizes the design choices and tradeoffs in iROS that follow from the consequences of the Volatility Principle and Boundary Principle.

Table 1: Summary of design decisions, their motivations and implications

Design choice	Reason	Benefit Realized
Levels of indirection, placing functionality in the comm server infrastructure	VP1,VP2	Failure isolation; client simplicity
Logically-centralized communication model	VP2	Evolvability: intermediation and snooping facilitate creating new behaviors and integrating new devices
Centralized implementation of communication model	VP2,BP1	Client simplicity; ease of policy management; higher likelihood of adoption due to architectural simplicity
Restart-based recovery using auto-reconnect	VP1	Ease of recovery, overall architectural simplicity
Use of beaconing and soft state	VP1	Simplicity of mechanism in failure resilience

3 iROS Functional Overview

iROS consists of three large subsystems, each of which is the subject of one or more separate publications. Since space does not permit us to repeat the details of their implementations here, we limit our description to the salient operational features that will be used to illustrate and quantify the tradeoffs referred to in the previous section.

The scenario in section 2 is representative of our experiences in iRoom for the past three years. We have found that the following functionalities are desired in interactive workspaces:

1. Dynamic application coordination, to enable each of the (originally standalone) viewer applications to reflect view changes and actions in the other viewers.

2. Moving data among displays or machines, possibly involving datatype transformation, to allow the viewers to run on any of the displays (group members' laptops or large touch screens) and views to be adapted to simpler devices such as PDA's.

3. Control of anything from anywhere, to allow remote control of the room's physical facilities and control of touchscreens from any laptop's mouse and keyboard.

We do not claim that the above is a complete set, but our experience confirms that these are necessary functionalities and they have allowed us to build a variety of useful applications to be easily written (in section 5 we try to give a sense of this breadth).

The programming model for iROS is one of ensembles of independent entities that communicate via message passing ("events") using a logically-centralized broadcast-like communication substrate. The entities are standalone applications or components whose behaviors can be linked by the exchange of events, or centrally controlled by emitting groups of events in response to a single user command. The basic message-passing mechanism that enables this coordination is called the EventHeap; on top of this we provide the DataHeap, a facility for moving data among heterogeneous devices using automatic datatype transformation, and ICrafter, a facility for on-the-fly, device-independent generation of human interfaces for hardware and software services or groups of services. We describe the salient features of each of these three subsystems.

3.1 EventHeap: Dynamic Application Coordination

The EventHeap [13] implements a coordination model based on tuplespaces [8] and forms the underlying communication infrastructure in iROS. The EventHeap implementation is client-server based, with the event storage and matching logic entirely implemented by the server to keep the functionality required on the clients simple. Applications can communicate with the Event Heap in one of three ways. First, new applications or applets written in Java, Python, or C/C++ can use communication libraries that provide primitives for posting events, querying for events, and subscribing to event streams. Second, Win32 applications that export COM API's can be wrappered for use in EventHeap applications. Third, a Java servlet that converts well-formed URL's and HTML form submissions into events allows the creation of HTML-based post and query interfaces to the Event Heap; examples of its use include Multibrowsing [15], which allows authoring of multi-displayaware Web content, and our original room-control application, which was implemented as a web page (allowing it to run it on handhelds for free).

Two important differences between the EventHeap and the traditional tuplespace model are the use of self-describing data and the garbage collection of events based on expiration times. An event is an unordered collection of name/value pairs; since fields are referenced by their (string) names, events are self-describing. In our system, one field is designated as the event type; entities agreeing on a particular event type are agreeing to the semantics of at least a subset of the remaining named fields. Components retrieve/subscribe to events using event templates, which contain precise values for fields to be matched and wildcards elsewhere. The EventHeap provides referential decoupling: the intended recipients of an event are determined by the contents of the event itself rather than being directly named (similar to intentional naming [2]), in that one or more fields of the event specify the desired attributes of the intended receivers.

Events are posted with expiration times, and expired events are periodically garbage collected by the server; later we show how this facility supports service advertisement and sidesteps resource-reclamation issues. Event expiration times are upper bounds, since events do not persist across EventHeap server failures. The EventHeap was used for application coordination in the scenario in section 2, as follows:

1. The construction site map allows the selection of various view points in the site and emits an appropriate view change event. (Although this viewer had already been written, it could have also been implemented as an HTML page with an imagemap whose click regions link to URL's directed to the Event Heap servlet, generating an event when clicked.)

2. The schedule viewer displays tables of construction site information and emits "change date" events as dates are selected. This is easy to implement using a third-party Visual Basic-to-Java bridge that allows two-way integration between EventHeap clients and Microsoft Excel, registering a VB handler that is invoked when a cell is clicked.

3. The 3D wireframe viewer responds to "change view" and "change date" events emitted by the other viewers. It also provides its own UI for manipulating the model; actions on this UI cause the corresponding events to be emitted and picked up by the other viewers.

Modifying the original standalone viewers to use the EventHeap required no more than about 100 lines of code each. When any of the viewers are run standalone, the events it emits are simply ignored, and because of Event Heap garbage collection, they eventually expire rather than accumulating until consumed. If more than one viewer is active, they coordinate as expected. If a particular viewer is launched on multiple displays (whether laptops or wall screens), because of the referential indirection provided by the Event Heap, all instances can react to and generate display events. Similarly, because of referential indirection, the expected coordination behavior is observed regardless of which machine(s) the viewer(s) are launched on.

3.2 DataHeap: Moving Data

The Data Heap provides type-independent and location-independent storage of large and semi-permanent data in an interactive workspace. A datatype transformation system [16] uses a set of dynamically composable data transformers to convert data among arbitrary formats. Automatic data transformation can often be imperfect/lossy, but since the transformed data is usually meant to be displayed for users (rather than processed by other programs), transformed data is still useful despite this limitation.

The Data Heap stores the actual data on a WebDAV [1] server and the corresponding metadata (including the datatype) in a fast in-memory XML database. Using the Event Heap, data producers indicate their desire to store a document and its associated metadata, and consumers query for metadata and indicate which formats they can accept. The Data Heap responds to consumer queries by dynamically instantiating a chain of transformation operators to convert the data to one of the acceptable types. Hence, the Data Heap frees data producers from having to know in advance the capabilities of consumers, and thus from having to know who the consumers of their data will be. This property is essential in a multi-platform ubicomp environment in which not all clients agree on a canonical set of data formats.

The DataHeap supports the meeting scenario as follows. A meeting participant uses a DataHeap client application on her laptop to give the model data a name and move the data from her laptop to the DataHeap server in its native (viewer-specific) format. The PDA user's viewer queries the Data Heap for the metadata using that name, specifying the data formats it can handle; the DataHeap server automatically transforms the model data to the requested format before returning it.

3.3 ICrafter: Control Anything From Anywhere

ICrafter [20] is the iROS UI-generation framework. We refer to any controllable hardware or software entity (room physical plant, software application, etc.) as a service. ICrafter provides a mechanism for services to publish short-lived "beacon" events to announce their presence; service discovery is accomplished by querying the EventHeap for available events whose type field matches `Beacon`. The beacon events contain a description of the service's available method calls in an XML-based markup language called SDL. (Current efforts in the Web community such as WSDL are similar, but were immature or nonexistent when iROS was designed.)

To obtain a UI for a given service, the user makes a request of the interface manager (IM), which is a standard service provided by iROS. The IM selects one or more *UI generators* to produce a UI from the service description embedded in the beacons. A UI generator may be specific to a type of service (e.g. projector controller), specific to a device's UI toolkit (e.g. HTML, Java Swing, VoiceXML), both, or neither; the IM tries to select the most specific generator possible from its repository of (hand-written) generators, and as explained in [21], can also be configured to automatically search a global repository. The level of indirection represented by the IM reduces the barrier to adding a new service or device: if a new service for which no service-specific generator is available for the locally-supported UI toolkits, the IM can automatically try to acquire a service-specific generator, and if it is unable to find any, it can fall back to using a service-generic generator that automatically generates a functional (if aesthetically clumsy) UI directly from the SDL. Similarly, if a device supporting a new UI toolkit is added, the IM can automatically acquire service-generic or service-specific UI generators that target this new UI toolkit. The sophistication of the IM means that services do not have to know in advance what kinds of devices will be requesting their UI's; analogously to the Data Heap, this lowers the barrier to integrating a new service or device into iROS.

ICrafter supports the lighting control illustrated in the meeting scenario as follows.

The user launches a simple Java applet client on her laptop that uses Java Swing widgets to display all available controllable services. Generating this list involves discovering the available services by querying the Event Heap for the available service beacon events. When the user asks the IM for a UI to the "lights" service, the IM searches the local generator repository for a lights UI generator tailored to Java Swing capable clients. Such a generator consists of XML-like markup describing the widgets, interspersed with script code; the script code is used to tailor the UI to the particular installation, e.g. by replacing low-level device names such as "light0" with more meaningful names such as "Main Overhead Lights". The generator executes the scripts to produce markup that can be rendered directly by the requesting client. Thus this level of indirection also enhances portability across workspaces.

4 Supporting Evolvability and Robustness

In this section, we describe and evaluate the key design choices in iROS relative to the functionality of the three subsystems described. Since iROS is a software infrastructure for a human-centered system, definitions of things like "acceptable" performance must be based on human interaction times. A study by Miller [18] identifies the following thresholds for R, the time it takes for the system to respond to a user's command:

- $R > 100$ms: the illusion of "instantaneous" response time is lost; user perceives the system as sluggish.
- $R > 1$sec: the user's thought process is interrupted and the delay is perceived as obtrusive.
- $R > 10$sec: the user becomes distracted from the task at hand and will start to work on other tasks while she waits.

We will use these values to establish thresholds for "adequate" performance in various cases.

4.1 Levels of Indirection for Evolvability

iROS makes frequent use of a level of indirection. In each case, the level of indirection adds latency but results in better failure resilience or ease of integration/evolvability.

Two examples illustrate the failure resilience benefits of communication indirection through the Event Heap:

1. Components communicating over tuplespaces do not have direct connections between them (referential decoupling). As a result, applications are not prone to failures due to disruptions in the connections. Of course, the connection between a component and the EventHeap server can also fail, but the EventHeap client library handles this failure, as will be explained in section 4.4. In general, the tuplespaces model encourages developers to write applications in a loosely coupled manner (even though it is still possible to create a tightly coupled application atop tuplespaces).

2. Since events persist until expiration or an Event Heap failure, communicating components often need not be up at the same time (temporal decoupling). That is, a transient failure in the receiver may be masked if the receiver restarts before an event directed to it expires.

With respect to evolvability, we may ask: what is the minimum amount of work required to integrate a new service or device into iROS? According to our three observed functionalities, the new device or service must be able to coordinate with other entities, participate in moving data, and allow itself to be remote controlled:

1. Since the event matching and buffering is entirely in the server, adding Event Heap communication capabilities to a client is simple. The full-featured Java client library has 740 semicolons, and the EventHeap-to-Web servlet previously mentioned allows simple Web pages and forms to both generate and query for events.

2. To allow the device to exchange data with other devices, we add a DataHeap transformer between the device's native data type(s) and some subset of those already supported. For example, the Mural cannot display Microsoft PowerPoint presentations but can display JPEG images; we built a simple PowerPoint-to-JPEG transformer using PowerPoint's ActiveX API, and consequently when a user asks to display a presentation on the Mural, the alternative JPEG version is shown. Cross-platform applications can thereby share data without being modified. The transformer consists of 83 semicolons in Java (231 LOC) plus 46 lines of XML description; about half of this code is common code also used in other transformers. A slightly more complex example is the wrapper for the open source package ImageMagick, which handles conversion among a large number of datatypes; that wrapper is 64 semicolons (268 LOC) plus 180 lines of XML description.

3. Unlike other systems for network service UI's such as UPnP, Jini and Hodes et al. [10], iCrafter isolates UI selection and generation in a level of indirection (the IM) separate from clients and services. In section 3.3, we explained how this level of indirection reduces the barrier to adding new clients and services. The Java ICrafter service API was designed with an explicit goal to reduce programmer effort to create new services. As a baseline example, wrappering Microsoft Internet Explorer into a barebones ICrafter service (that only supports the "navigate to URL" method) using the ICrafter service API in Java requires about 20 semicolons of Java code; the ICrafter service API then uses Java reflection to automatically generate the SDL that will be advertized for this service. Without additional effort, basic HTML and Java Swing UI's are generated by the automatic UI generation facility in the IM (although custom UI's can be created with more effort, if needed).

In each of the above three cases, the presence of a level of indirection allows a new device or service to be integrated with a minimum of effort.

4.2 Logically Centralized Communication

The logically centralized, event-driven model of the EventHeap is well suited for our domain because several application coordination scenarios in interactive workspaces are "naturally" event-driven as illustrated by the scenario of section 2. In addition, the EventHeap offers non-obvious benefits for even request-response style interactions, because a centralized communication model facilitates *snooping and intermediation*. That is, since events are indirectly sent between applications (through the EventHeap), an intermediary can observe an event from a source and generate one or more events of different types in order to cause a desired action in a different receiver or receivers. For example, consider Multibrowsing [15], an iROS application that allows one to "send" Web pages or other documents from any display in the room to any other display (including laptops,

Table 2: Effort needed for various integration tasks

Integration task	Libraries used	Number of semicolons
Creating a barebones IE ICrafter service	Java ICrafter service API, Third-party Java-COM bridge	20
Allowing Mural to display PPT slides	DataHeap transformer API, Third-party Java-COM bridge	84 semicolons + 46 lines XML
Integrating 3d viewer into CIFE application	EventHeap Java API	100
Multibrowse-based custom room control web page	HTML, EventHeap servlet helper web site	One web form to fill on the helper site per multibrowse link

wall displays, and the Mural). Early prototype application developers had hardcoded the names of target displays in the iRoom, making their applications non-portable to other iROS installations. We exploited the ability to snoop and intermediate in *mbforward*, a simple intermediary that picks up multibrowse events with specified values in their 'Target' fields and automatically re-routes them to different machines by generating new events. Using this mechanism, we were able to use Multibrowsing demos originally hardcoded to the iRoom for demonstrations in other locations, without changing any of the original source code. This illustrates the benefit of intermediation for evolvability and portability. [12] compares the Event Heap in detail to other communication models, and concludes that although intermediation may also be possible in systems based on RMI/RPC or message passing if applications are carefully constructed, the shared tuplespace model makes the process much more elegant, low effort and straightforward.

The level of indirection from logical centralization also provides other benefits, some of which we return to in section 5, including ease of integrating new devices and connecting them to existing behaviors.

The centralized communication model also has the benefit of ease of ensuring application-level scope: since broadcast and discovery are implemented using the EventHeap, so their scope is the set of entities communicating with that same EventHeap. This is why we have not implemented subnet multicast based schemes for discovery in ICrafter (as is done in other service discovery systems such as UPnP, Jini and SLP). We could use such schemes as heuristic aids to discover the locally available EventHeap servers (currently, users manually decide which EventHeap their applications should connect to), although ultimately the decision must often involve user input, e.g. when an itinerant remote user wishes to participate in a meeting at her usual workplace.

A disadvantage of the logically centralized communication model is limited scalability. But we show in the next subsection that at scales consistent with BP, we get sufficient performance.

4.3 Centralized Implementation For Client Simplicity

A centralized communication model could still be implemented in a distributed manner. We chose a centralized implementation because it is simple to build, debug and reason about (less than 1200 semicolons in Java), requires minimal functionality on each client, and avoids the need to redistribute client code when the implementation changes. In contrast, a distributed implementation is likely to be challenging since there is no obvious way to partition the write-intensive workload while preserving the ability to intermediate (all clients must potentially be able to see all events). Every client would have to implement event buffering, event matching, and expiration.

The downside of a centralized implementation is limited scalability. However, BP limits the desired scope of the shared communication mechanism to a single workspace; a reasonable estimate is therefore that the system only needs to scale to the order of a hundred simultaneously active clients, and as previously mentioned, a reasonable threshold for adequate performance is an event delivery latency of around 100ms. Usually, the iRoom has no more than 30 simultaneously active clients and the peak aggregate request rate on the server is no more than 30 requests/second. Under these conditions, the latency is only 10ms.

We also conducted experiments to test the limits of our server implementation with respect to the number of clients (figure 3) and the aggregate request rate (fig-

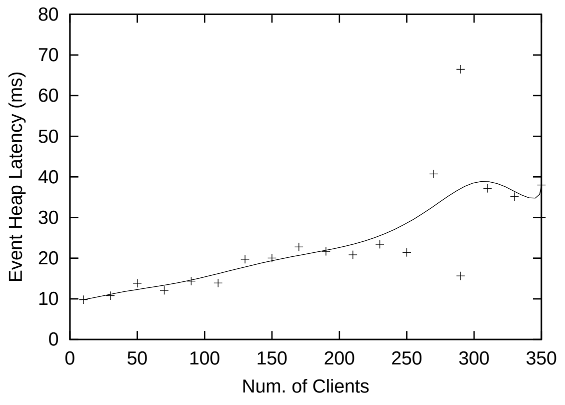


Figure 3: EventHeap latency vs. number of clients connected, with each client generating 100 events/sec. Each client generates a different event type.

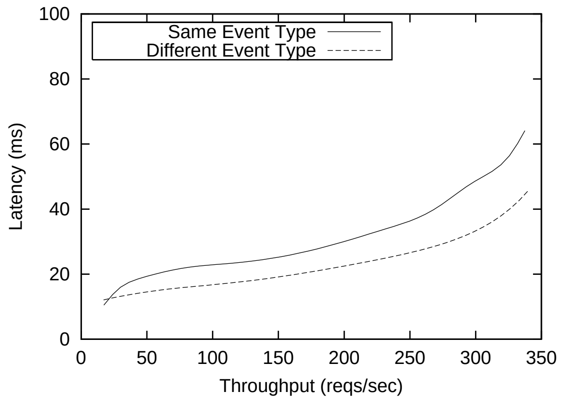


Figure 4: 100 clients simultaneously generating Beacon events; in the solid curve, probes of the same type (Beacon) were used to measure latency, while in the dashed curve, latency probes were of a type other than Beacon. Background traffic dominated by Beacon events is typical of an interactive workspace.

ure 4). Figure 3 shows that the latency is well below the 100ms limit for upto 350 clients even for our (non-tuned, straightforward) implementation of the shared communication model. Similarly, figure 4 shows that we achieve a latency well below 100ms for upto 350 requests/second. Consequently, we believe that the scalability of our server implementation is adequate for our setting.

Of course, centralized implementations are liable to be single points of failure. We address the potential single point of failure in the following subsection.

4.4 Restart-based Fast Recovery Using Auto-reconnect

A crash of the centralized EventHeap server is a single point of failure, which can in turn cause cascading failures as other components lose their connections to the server. We prevent this behavior as follows:

- As mentioned in section 3.1, events do not persist across EventHeap failures. As a result, we may lose some events during a crash, but the EventHeap can be recovered by restarting without any special recovery actions, and the restart time for the server itself is only 200 milliseconds.[1] The lost events can cause temporary disruption (e.g., a light control command will have no effect) but retrying the command after the EventHeap has recovered fixes the problem. (We decided against the significant additional complexity of supporting event recovery in the server, given that the server failure is a rare occurence.)

- Further, the EventHeap client library provides an auto-reconnect feature: connected applications detect EventHeap failure and they auto-reconnect when it is restarted.

- Some inconsistency is expected for a brief period following the restart of the EventHeap because all the built up soft state is lost in the crash. However, this state is automatically replenished in at most one beacon period after the clients reconnect.

Thus, the total time for recovery as perceived by the user is $TTR = T_{JVM} + T_{EH} + T_{RC} + T_B$, where T_{JVM} is the time to start the JVM, T_{EH} is the time for EventHeap initialization, T_{RC} is the time for all the clients to reconnect, and T_B is a beacon period. Typically T_{JVM} is between 1.5 and 2.5 seconds and $T_{EH} = 200$ms.

Consistent with Miller [18], we define "fast enough" recovery as 10 seconds, which is noticeable but unlikely to distract the user from the task at hand. Figure 5 shows the reconnect times for clients (T_{RC}) under varying values of the number of active clients N at the time the EventHeap fails. We suspect (but have not directly verified) that the differences in shapes among the curves are

[1]Placing the Event Heap startup command inside a `while(1)` loop recovers from JVM crashes; we are working on external monitoring to restart the Event Heap when livelock or thrashing is detected.

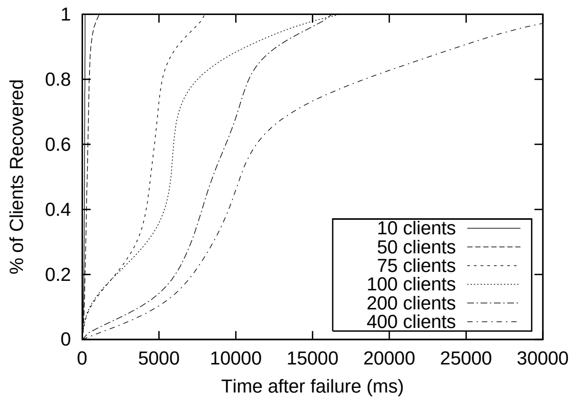


Figure 5: Speed of Event Heap recovery with different numbers of clients. The figure plots the fraction of clients successfully reconnected as a function of time.

due to TCP backoff, since all clients are simultaneously attempting to reopen TCP connections to the server.

From the figure, it may be inferred that for our typical operating parameters (less than 50 clients and a beacon period of 5 seconds), $TTR < 2.5 + 0.2 + 1.2 + 5$, i.e., less than 9 seconds. (We discuss the choice of beaconing rates in the next section.) While the recovery time is currently adequate for our purposes, we are exploring techniques for further improvement.

The auto-reconnect feature plays a key role in enabling dependency-free restarts of failed components. Without this, we would need to restart *all* iROS components, as well as all iRoom services and applications, after an EventHeap crash. (In fact, we had this problem in an earlier version of the EventHeap based on IBM TSpaces [28].) Finally, our failure handling strategy is simpler (and therefore, arguably more reliable) because it exploits the simplicity of the centralized implementation.

We do not argue that this is the *only* recovery method needed in an interactive workspace—it does not handle deterministic failures, such as a pathological event that always crashes the Event Heap, or hard failures, such as a persistent hardware failure on one of the machines. But it does handle a wide variety of transient failures, and we have verified from experience that most observed failures of iRoom software are in fact transient and curable through restarts.

4.5 Beaconing and Soft State for Dynamism

As explained earlier, services in iROS advertise their presence and other runtime information with periodic beacon events. The expiration time of a beacon event is set to twice the beacon period. Beacons help in better handling of partial failures (of the EventHeap and the other components) and dynamism:

1. “Stale” beacon events associated with components that failed or left the workspace will eventually expire and other components will detect their absence (after at most 2 beacon periods).

2. If the EventHeap server itself is restarted following a transient failure, all pending events, including service beacons, are lost. Use of beacons ensures that the service availability state can be built up again within 1 beacon period. Without beaconing, a transient failure in the EventHeap would have a cascading effect, since other services would also be perceived as having failed after the EventHeap is restarted. Thus, beaconing and auto-reconnect together prevent a transient Event Heap server failure from causing cascading failures.

While beacon-based soft state helps with failure resilience as shown above, it hampers scalability due to the increased load on the server. The scalability-robustness tradeoff associated with soft state is well known in the Internet systems community [22]. In our case the tradeoff can be captured smoothly with the beacon rate parameter. As the beacon rate increases, perceived consistency in the face of failures improves, but the overall performance of the system degrades due to the increased traffic. The following graph shows how the performance of the EventHeap degrades as the beaconing rate increases.

The results in figure 4 show that we can easily sustain a beacon period of 1 beacon/second for a hundred services. We conservatively set the beacon period to 5 seconds in iRoom, implying that inconsistencies can last for up to a maximum of 10 seconds.

5 Experience

iROS is a real system in daily use by multiple groups of non-systems researchers. Neither iROS nor the new applications being developed are bug-free, but the simple failure resilience mechanisms provided make recovery straightforward even for non-CS experts, and therefore encourage prototyping. Here we try to give a sense of what has and has not worked well in our experience.

5.1 Evolvability and Ease of Integration

Our experience in this area has been extremely positive due to the mechanisms described in section 4. The level of indirection provided by the DataHeap has made data sharing across platforms much easier: an enhanced version of the lights-and-projector control application allows drag-and-drop of URL's, images and documents for display on the touchscreens and Mural, using the DataHeap for transformation when necessary and allowing a natural extension of the multiscreen display mechanism across platforms.

The ability to "glue" applications together using the EventHeap is exploited in SmartPresenter, an application that choreographs the behavior of several independent copies of Microsoft PowerPoint to allow all the touchscreens to be used simultaneously in a presentation. The EventHeap's support for intermediation even allows a user in the audience to passively observe the presentation on her own laptop, simply by running a SmartPresenter client that snoops on events to any desired touchscreen.

Intermediation has also made it easy for us to integrate new devices and connect them to existing behaviors. For example, we have built about a dozen EventHeap-enabled wireless buttons [4] that can generate "button pressed" events containing the button's id. We created a simple GUI-based application that allows end users to map an iButton event to actions available from iRoom services. A generic intermediary uses these mappings to translate iButton events into service control events, the result being easy creation of physically-activated "macros" without any user programming. Although we do not support "destructive" intermediation in which intermediaries consume events before anyone else can see them, we have found that supporting only "additive" intermediation is useful in many situations.

The use of beacons facilitates the construction of applications that must respond to dynamism as they are running. For example, PointRight [14] allows any user to remotely-control a single mouse pointer across all iRoom screens, including laptops in the room; it uses the EventHeap to detect when displays join or leave the room infrastructure, and modifies the display topology on the fly. A similar technique is exploied in iPong [4], a simple Pong-like game in which the playing field is a dynamically-varying set of displays.

5.2 Other iROS Applications and Deployments

Regular group meetings in the iRoom routinely use most of the applications above. iROS has also been deployed in several other non-CS environments. The Center for Integrated Facilities Engineering (`http://cife.stanford.edu`), whose work is the basis of the scenario at the beginning of this paper, uses iROS to prototype large-site construction management applications in their lab. The Program in Writing and Rhetoric has deployed iROS on an experimental basis to prototype teaching strategies for collaborative writing. iROS is expected to be the base technology for new distance-learning classrooms to be completed in 2003; one of our developers has deployed a desktop-scale version of iROS to assist in development of applications for that environment. Although the deployments have been far from perfect (we describe some problems below), iROS has been sturdy under a variety of conditions of use by people other than its creators.

5.3 Ongoing Work

The downside of handling failure as a common case is that sometimes a true failure can be difficult to track down. A drawback of the referential decoupling we exploit for application coordination is that it is not meaningful to talk about end-to-end delivery semantics of messages, since the sender does not know in advance who the receiver(s) will be or whether there will be any at all. The most common symptom is the user issuing a command (e.g. turn on the lights) and seeing no effect. Our graphical Event Heap debugger (coincidentally also based on snooping and intermediation) is suitable only for sophisticated users, and some problems are actually due to hardware (e.g. malfunctioning X10 controller) rather than software.

We do not know of any comprehensive security model for collaborative workspaces. Inside the iRoom we rely on social conventions (e.g. it's rude to turn off the lights during a meeting); outside we rely on a firewall, although in keeping with the Boundary Principle we are moving toward controlling access to the EventHeap instead.

6 Related Work

6.1 Similar Approaches to Robustness

The intentional naming system (INS) [2] uses soft state protocols to maintain weakly consistent names across the name resolvers. This entails periodic name updates that cause scalability concerns, but in exchange provides robustness to dynamic environments. Similar to our use of application-level boundaries with the EventHeap rather than network-level boundaries, INS uses application-level resolution of query criteria (e.g., "anycast to the best printer" might name the least loaded printer rather than a printer on the same network).

The SNS/TACC scalable network server prototype [6] uses automatic start and restart for worker processes, making recovery from a worker crash the same as normal operation. Further, by using soft state for its load balancing, its load balancer's recovery procedure is also part of normal startup.

The Recovery-Oriented Computing project [19] presents evidence that as a result of Moore's Law and increasing system complexity, systems costs today are dominated by costs of downtime and ongoing maintenance. They argue that the former can be addressed by a renewed emphasis on design for graceful recovery (rather than strictly on failure avoidance) and that the latter can be addressed by designing systems for ease of administration and maintainability. We agree with these goals and have optimized our architecture to achieve them.

6.2 Other Ubicomp Frameworks

Surprisingly, not all ubicomp frameworks explicitly address the issue of evolvability and the resulting inevitable heterogeneity. One.world [9], GaiaOS [23], iLAND [26], and Jini [3] all require applications to be (re)written using specific languages and API's, and none provides for the integration of heterogeneous, whole-system building blocks.

Similarly, few ubicomp frameworks explicitly address the robustness problem. A noteworthy exception is one.world: their "programming for change" philosophy requires that applications be prepared to re-acquire lost resources at any time, and their API's decouple applications from resources to make this possible. Applications must include special code to recover from operations that fail due to having lost a resource, but their framework clearly separates those operations from ones that cannot fail, and they provide support for handling several common cases. Further, their state-encapsulation boundaries in general isolates the failures of components or applications.

7 Summary and Conclusions

iROS addresses three challenges: it provides support for abstractions needed by a large class of demonstrably-useful ubicomp applications; through appropriate design choices, it provides the failure resilience and ease of evolvability and maintainability required of ubicomp; and it accomplishes these goals with an architecture that is simple to understand and develop on. The ease of development and administration make iROS a useful off-the-shelf building block for ubicomp (non-systems) researchers. Although other ubicomp projects have addressed subsets of these challenges, none that we know of has successfully addressed all of them.

We do not claim that any single aspect of our solution—recovery behavior, scalability, or performance—represents a revolutionary improvement in and of itself. But we believe the combination of these in iROS represents the *simplest* solution that satisfies all the constraints, and we tend to agree with Gabriel [7] that simpler systems that are "good enough" are more likely to be adopted by future researchers and thereby improved over time. Indeed, we hope that others will improve on our efforts.

Ubicomp deserves the attention of systems researchers if we want to avoid creating brittle and hard-to-use systems. Systems building is about making design choices in the face of design constraints; we hope that our experience with iROS will help other systems researchers addressing comparable design constraints in making informed design choices.

For more information, to download iROS, or to see videos of iROS in action, please visit `http://iwork.stanford.edu`.

References

[1] Web-based Distributed Authoring and Versioning. Available at `http://www.webdav.org`.

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP-17)*, volume 33, pages 186–201, December 1999.

[3] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison Wesley, 1999.

[4] J. Borchers, M. Ringel, J. Tyler, and A. Fox. Stanford Interactive Workspaces: A Framework for Physical and Graphical User Interface Prototyping. *IEEE Personal Communications Special Issue on Smart Homes*, June 2002.

[5] W. K. Edwards and R. E. Grinter. At home with ubiquitous computing: Seven challenges. In *Third International Conference on Ubiquitous Computing (Ubicomp2001)*, 2001.

[6] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, St.-Malo, France, October 1997.

[7] R. P. Gabriel. LISP: Good news, bad news, how to win big. *AI Expert*, 6(6):30–39, 1991.

[8] D. Gelernter. Generative communication in LINDA. *ACM Transactions on Programming Languages and Systems*, pages 80–112, January 1985.

[9] R. Grimm et al. Systems directions for pervasive computing. In *Eighth Workshop on Hot Topics In Operating Systems (HotOS-VIII)*, pages 147–151, sep 2001.

[10] T. Hodes and R. Katz. A Document-based Framework for Internet Application Control. In *Second USENIX Symposium on Internet Technologies and Systems (USITS 99)*, pages 59–70, October 1999.

[11] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan. Distributed Rendering for Scalable Displays. In *IEEE Supercomputing 2000*, 2000.

[12] B. Johanson and A. Fox. Tuplespace-based Coordination Infrastructures for Interactive Workspaces. In Submission to Journal of Systems and Software Special Issue on Application Models and Programming Tools for Ubiquitous Computing.

[13] B. Johanson and A. Fox. The Event Heap: A Coordination Infrastructure For Interactive Workspaces. In *Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 02)*, Callicoon, NY, June 2002.

[14] B. Johanson, G. Hutchins, T. Winograd, and M. Stone. PointRight: Experience with Flexible Input Redirection in Interactive Workspaces. In Submission to 15th Annual Symposium on User Interface Software and Technology.

[15] B. Johanson, S. Ponnekanti, C. Sengupta, and A. Fox. Multibrowsing: Moving Web Content Across Multiple Displays. In *Third International Conference on Ubiquitous Computing (Ubicomp2001)*, 2001.

[16] E. Kiciman and A. Fox. Using Dynamic Mediation to Integrate COTS Entities in a Ubiquitous Computing Environment. In *Handheld and Ubiquitous Computing (HUC 2000), First International Symposium*, Sept. 2000.

[17] T. Kindberg and A. Fox. System Software For Ubiquitous Computing. *IEEE Pervasive Computing Magazine*, 1(1):70–81, January 2002.

[18] R. Miller. Response time in man-computer conversational transactions. In *Proc. AFIPS Fall Joint Computer Conference*, volume 33, pages 267–277, 1968.

[19] D. A. Patterson et al. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB/CSD-02-1175, U.C. Berkeley, 2002.

[20] S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *Third International Conference on Ubiquitous Computing (Ubicomp2001)*, 2001.

[21] S. R. Ponnekanti, L. A. Robles, and A. Fox. User Interfaces for Network Services: What, from Where, and How. In *Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 02)*, Callicoon, NY, June 2002.

[22] S. Raman and S. McCanne. A Model, Analysis, and Protocol Framework for Soft State-based Communication. In *Proceedings of the ACM SIGCOMM Conference*, Cambridge, MA, Sept. 1999.

[23] M. Roman et al. GaiaOS: An Infrastructure for Active Spaces. Technical Report UIUCDCS-R-2001-2224 UILU-ENG-2001-1731, UIUC, 2001.

[24] J. Saltzer, D. Reed, and D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[25] M. I. Seltzer and M. A. Olson. Challenges in Embedded Database System Administration. In *Proc. USENIX Embedded Systems Workshop*, Cambridge, MA, Mar 1999.

[26] N. Streitz, J. Geibler, and T. Holmer. Cooperative Buildings - Integrating Information, Organization, and Architecture. In *First International Workshop on Cooperative Buildings (CoBuild 98)*, pages 4–21, February 1998.

[27] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.

[28] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. TSpaces. *IBM Systems Journal*, 37(3), August 1998. Available at `http://www.almaden.ibm.com/cs/TSpaces`.