BALANCING ZERO-ADMIN
AND INCREMENTAL INTEGRATION
IN UBICOMP ENVIRONMENTS

Andy Szybalski
May 2004
Contents

1 Introduction 1

1.1 iROS ......................................................... 3
  1.1.1 iROS Applications ................................. 4
  1.1.2 Incremental Integration in iROS ................. 4
1.2 Challenges of Bringing Ubicomp to the Real World .... 5
  1.2.1 Classroom 2000 ....................................... 6
  1.2.2 Wallenberg Hall ..................................... 6
1.3 Zero-Admin ................................................. 7

2 iROS Manager 9

2.1 Recoverability ............................................. 9
  2.1.1 Which Applications Need Recoverability? .......... 10
  2.1.2 Restartability ....................................... 10
  2.1.3 Stability via Redundancy ............................ 11
  2.1.4 Monitors ............................................ 12
2.2 Configuration ............................................. 12
  2.2.1 Application Configuration .......................... 13
  2.2.2 Global Configuration ............................... 15
2.3 Installation ............................................... 15
2.4 Boundaries ............................................... 16

3 Case Study: TeamSpace 18

3.1 Goals ..................................................... 18
3.2 User Experience ......................................... 20
3.3 How TeamSpace Approaches Zero-Admin ............... 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.1 Recoverability</td>
<td>22</td>
</tr>
<tr>
<td>3.3.2 Configuration</td>
<td>22</td>
</tr>
<tr>
<td>3.3.3 Installation</td>
<td>24</td>
</tr>
<tr>
<td>3.3.4 Boundaries</td>
<td>24</td>
</tr>
<tr>
<td>3.4 Adaptations to iROS</td>
<td>28</td>
</tr>
<tr>
<td>3.4.1 Integrated GUI</td>
<td>28</td>
</tr>
<tr>
<td>3.4.2 Consistent Naming</td>
<td>28</td>
</tr>
<tr>
<td>3.4.3 Summary of Changes</td>
<td>29</td>
</tr>
<tr>
<td>3.4.4 Extensibility</td>
<td>29</td>
</tr>
<tr>
<td>3.5 Lessons</td>
<td>30</td>
</tr>
<tr>
<td>3.5.1 Mundane Requirements</td>
<td>30</td>
</tr>
<tr>
<td>3.5.2 Beyond Simple Restarts</td>
<td>31</td>
</tr>
<tr>
<td>3.5.3 Implications for Zero-Admin Workspaces</td>
<td>32</td>
</tr>
</tbody>
</table>

4 Future Work and Conclusions 34

Bibliography 37
Abstract

Ubiquitous computing technology has been shown to substantially enhance collaboration, but currently, such technology is generally confined to academic environments. Real-world ubicomp environments are likely to be incrementally integrated, or assembled piecemeal from a heterogeneous set of custom and off-the-shelf applications. The ideal incrementally integrated environment does not restrict which applications can be integrated.

However, such a complex environment poses challenges to real-world administrators. What is needed, then, is an environment that is as easy to administer as possible. We enumerate four criteria which define a zero-admin environment: recoverability, configuration, installation, and boundaries.

iROS is a incrementally integrated ubicomp architecture that is built to support dynamic and heterogeneous interactive workspaces, but which currently lacks the properties of a zero-admin ubicomp infrastructure. This paper presents iROS Manager, a lightweight daemon that starts, stops, monitors, and configures arbitrary applications. iROS Manager is a general model for achieving many properties of a zero-admin environment while still maintaining most of the flexibility of incremental integration.

We explore TeamSpace as a case study of deploying a public zero-admin interactive workspace based on iROS Manager. TeamSpace illustrates that to attain certain properties of zero-admin, some degree of incremental integration must be sacrificed. We expect that balancing the goals of zero-admin and incremental integration will be a key challenge in developing the infrastructure to bring ubicomp to the real world.
Chapter 1

Introduction

Since 1999, the Interactive Workspaces project has explored ways to enhance collaborative work with the use of ubiquitous computing technology. Much of this research has focused on the Stanford iRoom [9], a conference room augmented with large touch-screen wall displays and wireless input and output devices. Ubicomp environments such as the iRoom are characterized by a large number of devices—including servers, laptops, and PDAs—that dynamically enter and leave the environment. These devices are running a heterogeneous set of custom-built and off-the-shelf applications.

To support these needs, the iRoom makes use of a software infrastructure called iROS, a meta-OS that ties together devices with different operating systems, human-computer interfaces, and levels of processing power. iROS is built in such a way that there is a loose coupling between and among devices and applications. This enables fluid interactions among a changing set of heterogeneous components.

We refer to this approach as incremental integration: rather than building monolithic ubicomp environments all at once, designers create applications and behaviors by integrating existing large building blocks [9] over time. Integration may include coordinating the behavior of these building blocks through an external communication infrastructure [8], “peeling off” the human interface of applications so that it can be replaced with other modalities [15], or rapidly adding new behaviors or new modes of controlling existing behaviors through external or physical UI’s [2].

One key principle of incremental integration is that systems should minimize the effort of integration. Most basically, this means that application code should not need to be modified in order for an application to be integrated into a ubicomp environment. This
Chapter 1. Introduction

Figure 1.1: The Stanford iRoom (left) and the KTH iLounge (right) are interactive workspaces enabled by iROS technology.

is especially important because ubicomp administrators may not always have access to the codebase of off-the-shelf software such as Adobe Photoshop or Microsoft PowerPoint. Nevertheless, they still might want to integrate these applications into the environment. Beyond the modification of code, this effort of integration may also include installation and configuration procedure. The higher the effort of integration into an environment becomes, the less flexible the environment is.

In keeping with this idea, the iRoom consists of a set of loosely coordinated applications and devices that together form an integrated system. Over time, this architecture has enabled various new and useful behaviors to be incrementally added to the iRoom. iROS has also been deployed in other ubicomp environments such as the KTH iLounge (see Fig. 1.2) and several locations throughout the Stanford campus.

Road Map

This chapter describes iROS as a model of an existing ubicomp architecture being used in a research environment. It then explores the challenges of adapting ubicomp software to a real-world situation, as illustrated by previous work on Classroom 2000 and Wallenberg Hall. It then defines the properties of an ideal zero-admin ubicomp environment, or one that provides recoverability, requires minimal configuration and installation, and sets clear boundaries on interactions.

Chapter 2 presents iROS Manager, a daemon that manages the starting, stopping,
configuration, and installation of processes in an iROS environment. In Chapter 2, we describe how iROS Manager brings ubicomp closer to the ideal of zero-admin, including how it employs restartability to achieve recoverability. We also discuss the sacrifices to incremental integration that are inherent in the use of restartability. iROS Manager is presented as a model for building a zero-admin environment while maximizing ease of integration as much as possible.

Chapter 3 is a case study of TeamSpace, a public interactive workspace based on iROS Manager that has been deployed in Meyer Library at Stanford. We discuss how iROS applications were specialized for TeamSpace, and weigh the benefits and drawbacks of this specialization. We also discuss various mundane lessons that we learned while building TeamSpace.

Chapter 4 discusses future work and anticipates the challenges that lie ahead in balancing incremental integration with zero-admin.

1.1 iROS

Nearly all iROS applications in a workspace communicate with one another indirectly by sending and receiving events via a single server. This server, the Event Heap [8], provides a tuplespace with publish-subscribe semantics. An event posted to the Event Heap by one application is visible to all other applications connected to the Event Heap. Applications do not communicate with one another directly; this helps prevent failures caused by complex interdependencies between applications, and it makes application development easier. Event Heap client code is available in various languages for various platforms, so communication between applications is independent of the platforms on which the applications are running. Hardware devices that cannot communicate directly with the Event Heap—for example, Bluetooth, RF, or USB devices—send and receive events via an iStuff [3] proxy application.

Since all iROS applications in a ubicomp environment are connected to the same Event Heap, this Event Heap defines the boundaries of interaction within the environment. Almost always, these boundaries coincide with the physical boundaries of the environment. This satisfies the Boundary Principle [13], which states that the boundaries of interactions within an environment should correspond to the physical boundaries of the environment, and that these boundaries should be apparent to and explicitly changeable by users.
1.1.1 iROS Applications

Two of the most commonly used iROS applications are PointRight and MultiBrowse. PointRight allows pointer input to be redirected from one machine in a ubiquitous computing environment to another. Once a pointer topology is specified, a pointer can be moved off the edge of a screen on one machine, and after the pointer rolls over a brief “speed bump,” it will appear at the corresponding edge of an adjacent display, accompanied by a flash to attract the user’s attention. PointRight allows a user to easily interact with objects on different displays. While its functionality is similar to that provided by Windows and Mac OS X on machines with multiple screens, PointRight actually bridges between multiple machines instead of just multiple screens.

MultiBrowse consists of two complementary applications, MultiBrowseDroplet and MultiBrowseService. The former provides a GUI for sending files and links to any machine in the workspace. The latter a daemon that runs on every machine in a workspace and handles the receiving of files and links. In combination, they make it easy for users to share files and information during a meeting or while working collaboratively.

1.1.2 Incremental Integration in iROS

In the case of both PointRight and MultiBrowse, an instance of the application on one machine must communicate with an instance of the same application on another machine.

Since PointRight and MultiBrowse are only communicating between instances of the same application, they can use any event format as long as it is consistent. In other words, no interapplication integration is needed. A more complex situation arises when applications need to communicate with one another.

Specifically, some iROS components are not useful in themselves, only in combination with other components. For example, the iRoom contains several wireless buttons which, when pressed, put a ButtonDown event on the Event Heap (via a proxy application). Suppose a user wants a sound to play in the iRoom whenever a button is pressed. There is an iROS media player application that can play sounds; however, it only knows how to respond to PlaySound events. A similar situation exists for other desired interactions such as turning on the iRoom lights or projector with a wireless button, or controlling the paddle in iPong using a wireless joystick or slider.

Phrasing the problem in more general terms, iROS dictates no standard event formats,
Figure 1.2: Patch Panel provides a generic event translation service to enable communication between applications and devices that have no a priori knowledge of each other’s event formats. (Note: The Patch Panel is actually a single application; it is represented here as two objects for visual clarity only.)

so in order for Application A to respond to the events of Application B, it must have a priori knowledge of Application B’s event formats. Since this is not always practical, iROS provides a service called the Patch Panel [2] to perform arbitrarily complex intermediation between event formats. The Patch Panel allows a user or administrator to set up a mapping $A \rightarrow B$, so that whenever an event of type $A$ is placed on the Event Heap, the Patch Panel translates it to an event of type $B$.

Returning to our example, we can configure the Patch Panel with a mapping $\text{ButtonDown} \rightarrow \text{PlaySound}$. Subsequently, whenever a $\text{ButtonDown}$ event is posted by one of the wireless buttons, the Patch Panel will translate the event to a $\text{PlaySound}$ event that will be recognized by the media player application. This is the behavior that was desired by the user. This incrementally integrated approach to interapplication communication makes iROS more flexible than tightly integrated systems such as Jini[17] or SpeakEasy [6], which more rigidly specify event formats.

1.2 Challenges of Bringing Ubicomp to the Real World

A few projects have deployed ubicomp environments in “real-world” situations. In most of these cases, expert developers have been on hand to address problems, so the systems
cannot be called zero-admin. However, they highlight some challenges associated with bringing ubicomp to the real world.

1.2.1 Classroom 2000

The Classroom 2000 project [1] deployed a system that captured live video and slides during a class and enabled students to access the captured information during or after lecture. Students reported that Classroom 2000 enhanced their educational experience. However, there are a number of obstacles to widespread adoption of systems such as Classroom 2000. Foremost, all Classroom 2000 applications were custom-designed, which the developers found a strain on time and resources. They noted that an incrementally integrated approach would be preferable.

In addition, the Classroom 2000 environment had to be closely monitored by the developers. Despite this fact, initial tests two of ten lectures were not recorded due to software failures. Clearly, in the real world, it is unrealistic for a team of expert developers to be available to fix occasional system failures. Instead, ubicomp environments must maximize availability and minimize workload for administrators.

1.2.2 Wallenberg Hall

Another lesson in real-world ubicomp environments comes from Stanford’s experiences in Wallenberg Hall. The Interactive Workspaces group enhanced two classrooms in Wallenberg Hall with a subset of iROS technology, including PointRight [10] for pointer redirection and MultiBrowse [11] for sending files and web links between computers. The computers in the environment were connected via a central Event Heap server.

Unfortunately, at the time, the installation of PointRight required the editing of configuration files to specify a pointer topology, which could only be done by an administrator well versed in iROS. Since the software had to be installed by an expert, students could not install iROS on their laptops and then use them to interact with the room. Instead, students used preinstalled and preconfigured student workstations. This fixed set of computers was always connected to the room’s Event Heap and always running PointRight and MultiBrowse.

We can look at this situation as a problem of incorrect interaction boundaries. As in the iRoom, the boundaries of interactions within a room are defined by the room’s Event
Heap. However, the students were unable to use their own laptops when they wanted to; in other words, the boundaries of the environment were too small. Furthermore, students who were unaware that their workstations were connected to the Event Heap would accidentally mouse off the top of their laptop screens. PointRight would cause their pointer to appear on the overhead screen, often disrupting the instructor’s pointer. In these cases, the boundaries were too inflexible, because students wished to manually remove themselves from the virtual environment but were unable to do so. These problems illustrate the importance of the Boundary Principle [13].

1.3 Zero-Admin

The lessons learned from the experiences of Classroom 2000 and Wallenberg Hall can be divided into two categories.

1. **Incremental integration:** The Classroom 2000 system was unable to incorporate off-the-shelf or previously-developed software. This placing excessive burden on developers, because the system could not be incrementally integrated; it had to be developed from scratch, all at once.

2. **Ease of administration:** Classroom 2000 administrators had to manually restart software after a failure. The imperfection of this system resulted in data loss for two lectures. In Wallenberg Hall, complications in the installation and configuration (including specification of boundaries) marred the usefulness and flexibility of the system caused problems. In both of these cases, a situation that was easier to administer would have proved more effective.

The size of the second category above indicates that there is more to deploying a real-world ubicomp environment than incremental integration. As ubicomp environments move from the research lab to the real world, administration will be performed by tech support staff with limited resources and limited familiarity with ubicomp technology. Thus, it is important for ubicomp environments to be as easy as possible to administer. We will define the **zero-admin** ubicomp environment as one with maximum ease of administration, specifically:

1. **Recoverability:** The environment is self-managing with respect to common transient
hardware and software failures. Administrators do not need to manually address crashes.

2. **Configuration**: Manual configuration is unnecessary; however, when desired, it can be done from a single location.

3. **Installation**: The architecture can be quickly and easily installed on a client computer, so that it is not limited to a fixed set of machines.

4. **Boundaries**: Users can never accidentally interact with the wrong environment, in accordance with the Boundary Principle[13].

These concerns must be balanced with the strategy of incremental integration in order to achieve a system that is easy to administer, built on an architecture that is reusable and not bound to a specific architecture or usage scenario. This framework defines our goals for the TeamSpace project.
Chapter 2

iROS Manager

We have claimed that order to adapt iROS technology to the real world, it will need to exhibit the properties of a zero-admin environment: recoverability, ease of configuration and installation, and consistent interaction boundaries. Furthermore, by examining the way physical environments evolve, we can see that software in real-world environments will be deployed piecemeal [16] Thus, we must minimize the effort of integrating a new application into the environment.

iROS Manager is presented as a model for adapting an incrementally integrated ubicomp environment such as iROS to meet the needs of a zero-admin, real-world situation. iROS Manager is a daemon that controls an heterogeneous set of applications in order to provide recoverability and decrease the effort of integration. In order to control an application, iROS Manager needs to have information about how to start, stop, monitor, restart, and configure it. As we will see, iROS Manager’s needs to specify application behaviors are often at odds with the need for incremental integration.

In this chapter, we will review each of these zero-admin properties as they were defined in Sect. 1.3, explaining how iROS Manager does and does not fulfill the properties, and at what cost.

2.1 Recoverability

The purpose of recoverability in a zero-admin environment is to ensure that common hardware and software crashes require no intervention by a human administrator.
2.1.1 Which Applications Need Recoverability?

In addressing the problem of how to provide recoverability, one might be tempted to say that recoverability is important only for infrastructure applications, and not for user-level applications. However, with the Patch Panel (see Sect. 1.1.2) facilitating complex interactions between arbitrary applications, the line between the two types of software is blurred: not only do user-level applications rely on infrastructure applications such as the Event Heap, they are also relied upon for services that are critical to the smooth operation of the environment. For example, without PointRight (see Sect. 1.1.1), a user cannot use his laptop mouse and keyboard to control other user-level applications such as Microsoft Word. And applications such as the iROS media player (see Sect. 1.1.2) provide services that may be used by infrastructure components to communicate important state to the user.

We could require developers or administrators to specify to iROS Manager the nature of services provided by each application, but this would preclude the type of rich interaction enabled by the Patch Panel. We deem this to be an unacceptable blow to incremental integration.

Without this information, then, we must assume that every application that connects to the Event Heap may prove some critical service. Consequently, it is not enough for iROS Manager to maximize availability of some fixed set of infrastructure applications; rather, it should maximize availability for all applications. For this reason, iROS Manager must control at least all applications that communicate with the Event Heap. Other applications that are deemed critical can also benefit from recoverability.

2.1.2 Restartability

Thus, our goal for iROS Manager is to provide recoverability for any application that interacts with the ubicomp environment. One approach to achieving this goal is to design applications to be restartable at any time. This makes recovery the same as normal restart and provides a simple and safe mechanism for attempting recovery [4]. Restartability has been successfully exploited in various fields as a cost-effective way to attain high availability with complex and unpredictable software. The distributed nature of ubiquitous computing makes it an ideal candidate for the restartability treatment. As we have described earlier [14], the iROS middleware (including the Event Heap client and server) is designed so that the server can be restarted with minimal interruption.
Although restart can also be used to “jump-start” applications whose performance has slowed, this is not appropriate for all applications. Many off-the-shelf applications are both brittle with regard to sudden restarts and characterized by intervals of unresponsiveness. For example, consider applying a complex filter to an image in Photoshop: the application may be unresponsive from the perspective of the OS for a few seconds, but killing it during that interval will destroy any work that has been done since the last save. In this case the best approach is simply to wait until Photoshop becomes responsive again. Similarly, some applications such as consumer-grade databases are safe to crash but take a long time to restart; in these cases restarting is not likely to improve availability. (For alternatives to restartability, see Sect. 3.5.2.)

Nevertheless, there are enough restartable applications to make restartability worthwhile in a ubicomp environment. To this end, iROS Manager selectively and hierarchically restarts failed components, thus maintaining a degree of availability without user intervention in the face of transient failures.

2.1.3 Stability via Redundancy

Since one stated goal of iROS Manager is to maximize the uptime of applications, the iROS Manager daemon itself must have high availability. To ensure this, the daemon actually consists of two very simple Java applications running in separate VMs. The first process, \textit{Starter}, handles only the starting and stopping of processes. The intelligence is located in the second process, \textit{Controller}, which handles communication with the Event Heap and processing of configuration files. The two processes are tightly coupled and communicate with one another via local UDP sockets.

The Starter and Controller processes periodically send “heartbeat” messages back and forth. If at any time one process detects that its counterpart is no longer running, it creates a new instance of its counterpart in a new VM. Likewise, if one of the processes detects a second instance of the same process, it commits suicide. This two-process model improves the uptime of the iROS Manager daemon by ensuring that there is always exactly one Starter and one Controller running (except when they are explicitly shut down by the user or in the unlikely event that they crash simultaneously).

For convenience, we will subsequently refer to the daemon as a single entity; however, by splitting it into two processes, we can trust it to be reliable enough to monitor and restart other applications when they fail.
2.1.4 Monitors

In order to take advantage of an application’s restartability, iROS Manager must determine whether an application is functioning properly, or alive. A monitor is a Java class that iROS Manager uses to determine whether an application is functioning properly. In most cases, this is accomplished by polling the application or listening for a beacon.

By default, iROS Manager uses a class called ApplicationMonitor, which simply checks whether the application’s process is still alive. Ideally, the application is designed to be crash-only [5]; in other words, instead of entering an inconsistent state, it always terminates. In the real world, however, software often misbehaves without terminating, so ApplicationMonitor is not always sufficient. What we want is a true measure of whether the application is functioning properly.

However, different applications have different definitions of “functioning properly”; thus, no one monitor will suffice for all applications. For this reason iROS Manager allows a programmer to subclass ApplicationMonitor to create a custom monitor for a particular application. The name of this custom monitor class is designated by an application’s configuration file (see Sect. 2.2.1).

Fig. 2.1 depicts a hypothetical iROS Manager configuration involving multiple custom monitors. As a real-world example, we have written a custom Event Heap monitor. Every three seconds, it posts a MonitorEvent to the Event Heap and then listens for that same event. If the time to complete this ping exceeds 1.5 seconds, the ping is considered to be a failure. If there are two consecutive failed pings, the Event Heap server is considered dead, and iROS Manager restarts it.

The Event Heap monitor demonstrates how iROS Manager can be used not only for crash detection but also for software rejuvenation—the process of restarting software to counteract the gradual degradation of the program’s state over the course of a session. The Event Heap monitor has been deployed successfully in the iRoom for almost a year. If the Event Heap server crashes or becomes sluggish, it is detected within 6 seconds (two pings) and restarted in another 5 seconds (the startup time of the Event Heap server).

2.2 Configuration

To have control over killing and restarting an application, iROS Manager must have control over starting it. First of all, this means that iROS Manager must know the location of the
application. Thus, for an application to be controlled by iROS Manager, it must be located in iROS Manager’s designated Software directory.

Furthermore, different applications must be started in different ways: some require parameters to be passed as command line arguments or as environment variables. These parameters may include the address of the Event Heap server for the environment—which will be the same for all applications—or the classpath for a Java application—which depends on the individual application.

Clearly, iROS Manager needs to know how to set any variables that are necessary to start an application. These variables must be specified in a config.ini file in the same folder as the application. This file tells iROS Manager what it needs to know in order to start the application. It is useful to look more closely at this file in order to further explore the capabilities of iROS Manager.

2.2.1 Application Configuration

Below is an abridged example of a config.ini file for MultiBrowseService.

---

Figure 2.1: A hypothetical iROS Manager configuration. Each custom monitor sends and receives information from its application to determine if it is alive. It shares this information with the local instance of iROS Manager, which in turn shares it with remote instances.
The file contains several “magic” variables which tell iROS Manager what to do with the application:

- The `APPLICATION` variable specifies what string will be executed on the command line in order to start the application. It references the environment variables `MACHINE_NAME` and `EHEAP_SERVER`, which are set in a global configuration file called `global-shared` (see Sect. 2.2.2).

- If an application’s configuration file contains the line `MODE = autostart`, that application will be started when iROS Manager is started.

- The line `KEEP_ALIVE = true` tells iROS Manager to restart this application if the monitor detects that it is malfunctioning.

- The `MONITOR` variable specifies the class to be used as a monitor for this process. The class must implement a specified interface which is part of the iROS Manager package. If this line is not present then the default `ApplicationMonitor` will be used.

Beyond these “magic” variables that are used by iROS Manager, the `config.ini` file can be used to set arbitrary environment variables before launching the application:

- In the example, `MB_OPTIONS` is one such environment variable.

- Also, paths can be concatenated to environment variables such as the `CLASSPATH` variable using the `+=` operator.
2.2.2 Global Configuration

Properties that are consistent across all applications for a ubicomp environment are defined in a configuration file called `global-shared`. This has the benefit that an application’s `config.ini` file can deal solely with properties of the application.

```plaintext
[global-shared]
MACHINE_NAME       = ?{Please enter a name for your machine}
EHEAP_SERVER       = teamspace-server.stanford.edu
FILE_SEPARATOR     = /
PATH_SEPARATOR     = :
SCRIPT_STARTER     = /bin/sh
SCRIPT_EXTENSION   = sh
```

Parameters in this file are accessible by all applications managed by iROS Manager. Note that this file contains environment-specific parameters, such as the Event Heap server, machine-specific parameters, such as the machine name, and OS-specific parameters, such as the extension and interpreter of shell scripts. On the first launch of an application, the user will be prompted for a value to satisfy each expression of the form `?{prompt}`. The value is stored in a preferences file but can be changed later.

2.3 Installation

One of our stated goals for iROS Manager is to provide zero-admin properties to incrementally integrated ubicomp environments in which hardware devices are dynamically being added and removed. For example, a user may bring in a laptop that does not yet have iROS Manager installed. To minimize the difficulty of integrating the laptop into the environment in this situation, iROS Manager is quickly and easily installable.

This helps avoid situations like the one in Wallenberg Hall (see Sect. 1.2.2), where the difficulty of installing software on new laptops artificially narrowed the interaction boundaries. It also highlights an advantage of iROS Manager over more industrial-strength systems such as Tivoli: difficulty of installation and licensing issues would preclude quickly installing Tivoli on a laptop brought into a zero-admin environment for the first time.
2.4 Boundaries

Because the Event Heap defines the boundaries of a ubicomp environment, it is very important that these boundaries be consistent and well-known. In the iRoom, where each application is configured independently, it is possible to have one iROS application connected to a different Event Heap than all the other applications in the room, and thus, logically part of another “room.” While this setup is occasionally useful for debugging or testing purposes, first-hand experience suggests that it causes undesired and confusing behavior.

Since all applications running under iROS Manager get the Event Heap setting from a single, global source (see Sect. 2.2.2), the boundaries of interactions are more consistent.

Moreover, the iROS Manager daemon running on a machine makes available information about that machine’s iROS configuration via the Event Heap. Thus, GUIs can be built to visualize the state of the environment. We have built two such GUIs. The first of these is iROS Menu (Fig. 2.2), which is intended for an intermediate to advanced user. When clicked, the menu displays the Event Heap to which the user is connected, and the running applications. It also allows the user to connect to, disconnect from, or switch the Event Heap server, or to start, stop, or restart applications.

The second of these GUIs, iROS Launcher (Fig. 2.3), is intended for an administrator,
Figure 2.3: The Launcher application is the most full-featured GUI for iROS Manager. It shows the status of all running applications in the room, and allows any installed application or group of applications to be started, restarted, or stopped. In this screenshot, the Launcher is showing that the MultiBrowseService application on the machine “iw-smartboard2” is down.

and provides a view of all machines connected to the Event Heap that are running iROS Manager. In addition, it lists the iROS applications running and installed on each machine. iROS Launcher provides a fine level of control over these applications, and allows groups of applications to be created that can be started, stopped, and restarted all at once.

Allowing the user to take control of iROS Manager using these GUIs fulfills an important part of the Boundary Principle [13]: consistent boundaries should be maintained by default, but users should have explicit control.

Note that no GUI is needed to run iROS Manager—it aims to make its work completely transparent to the user. Depending on the purpose of the environment, however, users may need to manually start or stop processes. In these cases, a custom GUI can be written to suit the specific needs of an environment.

We will see one such custom GUI in the next chapter, which is a case study of TeamSpace, a public interactive workspace based on iROS Manager.
Chapter 3

Case Study: TeamSpace

TeamSpace (Fig. 3.1) is a model for a public zero-admin interactive workspace. Currently one pilot TeamSpace exists in Meyer Library at Stanford University; more installations are in the planning stages. TeamSpace consists of two large public displays situated in front of a large table with room for four to five users. Students are encouraged to bring their laptop computers with wireless Internet access. The public displays are not touch-sensitive and the servers powering these displays are not connected to a public mouse or keyboard, so users must download and install the TeamSpace client software to their laptops in order to interact with each other and with the workspace.

The TeamSpace software (Fig. 3.3) runs on the servers and is installed on student laptops. It consists of iROS Manager, the TeamSpace GUI (a custom GUI for iROS Manager), and PointRight and MultiBrowse, two bundled iROS applications that run under iROS Manager.

3.1 Goals

TeamSpace aims to adapt components of iROS [14] such as MultiBrowse [11] and PointRight [10] to a public computer cluster setting. In doing so, we have tried to balance the goals of zero-admin and incremental integration.
Figure 3.1: Participants in a user study work collaboratively in the TeamSpace in Meyer Library. Photo: Richard Holeton

Figure 3.2: The TeamSpace installer application can be downloaded from the web.
3.2 User Experience

When a group of users enters a TeamSpace, a sign asks them to direct their laptops to http://teamspace.stanford.edu. There, they download and install the TeamSpace software, which is available for Mac OS X, Windows, and Linux. From the user’s perspective, TeamSpace is a single double-clickable application.

When a user double-clicks on the TeamSpace GUI application, entitled simply “TeamSpace,” a window appears as in Fig. 3.3. The window allows the user to specify the name of the local machine, and displays a “buddy list” of other connected users.

Once the TeamSpace GUI application is opened, a user can use the laptop pointing device to move the pointer off the top of the laptop displays using PointRight [10] onto the bottom of the main public display. While the laptop pointer is on the public display, a locking screen (Fig. 3.5) appears on the laptop display, informing the user that the pointer is currently on another display.

TeamSpace also allows the user to send and receive files, folders, and links from other users in the space using MultiBrowse [11]. On the local screen, the TeamSpace window contains a “buddy list” of the available remote machines. To send a file or folder to a remote machine, the user drops the icon on the machine’s name in the buddy list. The file is then opened in the remote machine’s default browser, generally giving the user a choice between opening or saving the received file. To send links, users can either drag the proxy icon in the browser’s address bar, or select the URL and drag it directly.

TeamSpace also provides an alternate method of sending files, folders, and links using the clipboard. If the clipboard contains a file, folder, or URL, clicking on a remote machine’s
CHAPTER 3. CASE STUDY: TEAMSPACE

21

Figure 3.4: The user sends a file, folder, or link to another machine by dragging the file’s icon and dropping it on the desired machine. Both the public displays and the displays of other users are available.

Figure 3.5: PointRight displays a “locking dialog” on the laptop display when the pointer is on one of the public displays.

name reveals a popup menu that allows the user to paste the object into that machine. In response to user requests, we are investigating ways to make the behavior of sending files and links more seamless using a shared clipboard or by integrating MultiBrowse with PointRight.

A user can interact with the environment using MultiBrowse and PointRight until the user closes the TeamSpace window. This defines the boundaries of the interaction. (These boundaries are discussed in detail in Sect. 3.3.4.)

3.3 How TeamSpace Approaches Zero-Admin

TeamSpace uses iROS Manager to manage a loose collection of separate applications. This approach allows TeamSpace to fulfill most of the properties of zero-admin as defined in Sect. 1.3.

However, this approach requires some sacrifices in incremental integration. Below we
revisit the criteria of a zero-admin environment one by one, noting how TeamSpace fulfills these criteria, and what sacrifices, if any, were made.

3.3.1 Recoverability

A running TeamSpace client installation consists of four running applications:

1. **iROS Manager**: (See Chapter 2) handles the stopping and starting of the processes below.

2. **TeamSpace GUI**: a custom GUI for iROS Manager. Manages the main TeamSpace window, and allows the user to send files and links to other computers in the TeamSpace using drag-and-drop.

3. **PointRight for Mac OS X or PointRight for Windows**: allows a user to mouse off the top of a laptop display onto a public displays. Since grabbing pointer control from the OS requires platform-dependent code, there are separate versions of PointRight for Mac OS X and Windows.

4. **MultiBrowseService**: receives and opens files and links from other computers in the TeamSpace.

In addition, one of the TeamSpace servers is also running Event Heap Server; this provides the Event Heap for the environment.

Because TeamSpace consists of multiple independent applications, iROS Manager can take advantage of the individual restartability of each application to achieve high availability. For TeamSpace GUI, PointRight, and MultiBrowseService, the default ApplicationMonitor is used for monitoring. This has proved to be more than sufficient, since none of these applications is especially crash-prone. The Event Heap server, which is prone to misbehavior, runs a custom EventHeapMonitor (see Sect. 2.1.4).

In this way, an additional burden is placed on incremental integration only for the Event Heap server, where a higher level monitoring is needed. We consider this an acceptable increase in the effort of integration.

3.3.2 Configuration

The above iROS applications, like most applications, generally have some application-specific state:
• **PointRight** reads the topology of the environment from a local file. This topology determines the connections between different edges of different screens. The topology file can be edited manually or using a GUI called Space Model Editor.

• **Event Heap Server** takes command line parameters specifying the port to use, as well as various performance settings.

• **Patch Panel**’s state is represented as a list of mappings in a local file. When Patch Panel is launched, it reads the mappings back from this file, and the file is updated while Patch Panel is open.

As the above list shows, these applications handle the configuration and maintenance of their internal state in completely different ways. This is to be expected, since they are independent applications, and this independence is desirable for the purposes of facilitating incremental integration. However, a system that maintains configuration in different ways in several independent locations is difficult to administer. The goal of TeamSpace is to pare configuration down to the absolute minimum, and to control any necessary configuration from a single location.

To eliminate unnecessary configuration, we can examine each application looking for parts of the configuration that remain constant. In the case of PointRight, since the physical configuration of the TeamSpace is known and constant, PointRight can use the same topology file all the time. Therefore, The user of TeamSpace does not need to configure PointRight at all.

The MultiBrowseService application requires only a few parameters, which it takes as command line arguments. These parameters include the user’s name and the Event Heap server. Both of these are known to iROS Manager: the TeamSpace GUI application asks for the user’s name at startup and passes it to iROS Manager, and the Event Heap server is a function of the user’s location, which is also specified at startup. Since iROS Manager has this information already, it can appropriately fill in the command line arguments to MultiBrowseService.

In this way, configuration can be unified under the umbrella of iROS Manager. In many cases, this eliminates redundancy: for example, PointRight takes the user’s name as an environment variable, and MultiBrowseService takes it as a command line argument. With iROS Manager, the name can be set a single time and passed appropriately.
In different ways, iROS Manager’s handling of configuration both detracts from incremental integration and adds to it. In order to start up an application with the appropriate command line arguments or environment variables, the application must be accompanied by a *config.ini* file specifying the appropriate parameters. This adds to the effort of integration. In a way, however, iROS Manager’s handling of configuration makes the environment *more* integrated, as it allows configuration to be handled from a single location. This dichotomy makes intuitive sense: for incrementally integrated applications to be more tightly integrated, the effort of integration will be greater.

### 3.3.3 Installation

As we have mentioned, the TeamSpace client software can be installed trivially in less than a minute. In addition to this, iROS Manager allows individual software components to be easily added to an existing installation of iROS Manager.

A software package running under iROS Manager must be accompanied by a *config.ini* file. If the software along with the configuration file are archived into a zip file, and given the extension *irz*, then iROS Manager will recognize it as a software package. Then double-clicking the file will automatically add it to iROS Manager’s configuration. This enables administrators to make *irz* packages downloadable from the web. Users can download additional or updated packages as needed.

This does not substantially increase the effort of integration; since *config.ini* files are already necessary, it is trivial to create an *irz* file from the package.

### 3.3.4 Boundaries

As stated in Sect. 3.2, a user can interact with the environment using MultiBrowse and PointRight until the user closes the TeamSpace window. More precisely, the boundaries of interactions within the TeamSpace are defined by the following conditions:

1. **Event Heap:** A TeamSpace installation contains a single Event Heap server through which all events are sent. The environment with which a user is interacting is determined by the Event Heap server to which the TeamSpace applications are connected on the user’s laptop.

2. **TeamSpace GUI:** When the TeamSpace GUI application is open, a color-coded status indicator clearly marks that the user is connected to TeamSpace (except in the
event of technical failures). When the application is closed, the user is disconnected from the TeamSpace.

3. Sessions: When a group enters the TeamSpace, they begin a new session. If these users leave the physical space but forget to close the application, then when a new group enters and starts a new session, the users from the previous session will be automatically disconnected. This prevents users from accidentally staying connected and interfering with TeamSpace interactions. Furthermore, a list of connected users is publicly visible in the TeamSpace window, so the system is transparent.

4. Physical Location: TeamSpace is designed to enable collaboration between users who are physically present in the space. While no special pains are taken to prevent outside users from connecting to the TeamSpace Event Heap, the TeamSpace is not useful to remote users.

This model satisfies the Boundary Principle requirements of visibility and explicit control. To implement this model, one must ensure that the application abides by each of the four conditions above. The following sections examine each of the conditions individually, as well as the associated implementation challenges.

(1) Event Heap

The first condition on TeamSpace’s interaction boundaries is that all applications are connected to the same Event Heap at all times.

It is trivial to ensure that this condition holds for MultiBrowseService and PointRight for Mac OS X. These applications simply take the Event Heap address as a command line argument, as illustrated by the MultiBrowseService configuration file (see Sect. 2.2.1). Since the Event Heap address will be the same throughout a TeamSpace session, all that is needed is to launch these applications with the correct Event Heap address. This is trivial.

However, it is not trivial to ensure that PointRight for Windows is using the same Event Heap as the other TeamSpace applications. This is the case because PointRight for Windows has its own way of deciding which Event Heap to use. PointRight for Windows can be passed an Event Heap as an environment variable, which is also trivial to accomplish with iROS Manager. However, by default it displays a dialog that allows the user to manually change the Event Heap, or to connect or disconnect. This dialog allows a meddlesome
user to create a situation where the interaction boundaries were inconsistent across iROS applications.

To prevent this situation, we modified the PointRight for Windows code so that it can be configured to always hide this window. We then modified PointRight’s `config.ini` to set the parameters that bring about this configuration. This way, we were able to make PointRight invisible and prevent users from changing the Event Heap.

We were able to make PointRight invisible because we had access to the PointRight codebase. If PointRight had been an off-the-shelf application, we would have had no choice but to accept the possibility of inconsistent boundaries. In this case, iROS Manager cannot fully realize both incremental integration and zero-admin. The only way to assure that the Event Heap is consistent across applications would be to decree that all applications that use the Event Heap must have a mode in which they are given an Event Heap address via a command-line argument or environment variable, and they use this same Event Heap until the application quits. This would be a significant erosion of the principles of incremental integration, as it places a heavy burden on software developers, and limits the array of software that can be integrated.

Furthermore, to achieve an environment that has truly consistent behavior across applications, it would be necessary for a similar decree to be made regarding other options such as a machine’s name, or sound volume, or whether links should open in a new window or in an existing window. The list is infinite. Unfortunately but unsurprisingly, it is impossible for a perfectly-integrated system to be built out of incrementally integrated components without defining extremely rigid specifications for those components.

(2) TeamSpace GUI

The second condition on TeamSpace’s interaction boundaries is that all applications are open if and only if the TeamSpace GUI application is open. It is trivial to ensure that this rule holds: when the TeamSpace GUI opens, it simply tells iROS Manager to launch all applications, and when it closes, it tells iROS Manager to kill all of the applications.

Since iROS Manager is designed to work with crash-only software, the only way it can quit an application is to kill it, using `kill -TERM` or `kill -9` on Linux or Mac OS X, and using `TerminateProcess()` on Windows. If software is not crash-only, these techniques might cause data to become lost or corrupted.

Most operating systems provide functions for asking applications to quit gently. If more
platform-dependent code were added to iROS Manager, it could call such a function and wait a few seconds before killing the application’s process as a last resort. In most cases, this acceptably mitigates the problem, although it is still impossible for iROS Manager to be sure of how long it should wait. For all current iROS software, which is crash-only, this is a non-issue. Still, trying a more gentle quit better achieves the ideal of incremental integration, so this technique will be explored in future work.

(3) Sessions

The third condition on TeamSpace’s interaction boundaries is that a user explicitly starts or joins a TeamSpace session with other trusted users, and he is connected to the TeamSpace until he explicitly leaves the session (by closing the TeamSpace GUI application) or until the session ends. The session can end for all users when one user clicks an “End Session” button on the TeamSpace GUI application on the public display. Alternately, inactive sessions are closed after a certain timeout period has elapsed.

When a session ends, all files saved during that session are wiped. In addition to providing clear interaction boundaries, this system also provides a level of security against casual snooping, and saves disk space.

It is trivial to implement this system as part of the TeamSpace GUI application. The ability to write custom GUIs for iROS Manager enables the addition of features such as this.

(4) Physical Location

The fourth condition on TeamSpace’s interaction boundaries is that a user is only connected when he is physically present in the TeamSpace. This is in line with the stated goal of TeamSpace—to enable collaboration among users who are physically present in the environment.

However, the current TeamSpace software currently takes no special pains to prevent users from remotely connecting to a TeamSpace. A mischievous user could MultiBrowse files and pages to public or laptop displays within the TeamSpace.

An naïve solution to this problem would be to exclude users outside the TeamSpace’s subnet; however, since many users connect to the TeamSpace using their laptops’ wireless Internet connections, this approach would exclude legitimate users. A partial solution would
be to display a “password” on the public screen which users must enter to join a session. Such a feature could be easily added to the TeamSpace GUI application.

Currently, however, we believe that social protocols will prevent most abuses. Unless they become a problem, it is preferable to keep the TeamSpace interface as simple as possible.

### 3.4 Adaptations to iROS

In the process of adapting MultiBrowse and PointRight to TeamSpace, we made a few changes to the code to “smooth over” some of the differences between the applications. These code changes were necessary to achieve acceptable levels of integration and zero-admin. These instances illustrate some of the limits of incremental integration.

One such instance was the modification of PointRight for Windows to prevent users from manually changing the Event Heap address (discussed in Sect. 3.3.4). Two other instances are described below.

#### 3.4.1 Integrated GUI

In iROS, there are separate applications for sending and receiving MultiBrowse events, called MultiBrowseDroplet and MultiBrowseService, respectively. In early versions of TeamSpace, each of these was launched as a separate application.

In the current version of TeamSpace, while MultiBrowseService is a separate application, the functionality of MultiBrowseDroplet is performed by the TeamSpace GUI application (see Fig. 3.3). This means that the GUI and MultiBrowseDroplet are not independently restartable; they can only be restarted together.

While it is easier to create a usable, integrated interface in a single application, restartability is best suited to a situation in which several processes can be independently stopped and started. This illustrates another practical limit of incremental integration and another situation in which the ideals of zero-admin and incremental integration are opposed.

#### 3.4.2 Consistent Naming

There are applications running called “PointRight” and “MultiBrowse,” but from the user’s point of view these are part of TeamSpace. To reflect this, TeamSpace is the only product
name that appears throughout the user experience. This was not an issue for MultiBrowse-Service, since it was invisible from the start. However, with PointRight for Mac OS X and Windows, and with MultiBrowseDroplet, the original applications prominently displayed their names and logos. We changed the code to instead display “TeamSpace.”

Without access to the codebase of these applications, we could not have expunged the original names. Obviously, there is a practical limit to incremental integration: no matter how much we “integrate,” it is impossible to change the basic functioning of a program without changing the code.

3.4.3 Summary of Changes

So far, we have noted a number of challenges associated with adapting iROS to a zero-admin environment. Table 3.1 summarizes the changes and additions to the iROS codebase that were made to accommodate TeamSpace.

Table 3.1: Adapting iROS applications for a zero-admin environment

<table>
<thead>
<tr>
<th>Application</th>
<th>Challenges</th>
<th>Changes Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>PointRight for Windows</td>
<td>Allowed user to change Event Heap, creating multiple inconsistent boundaries</td>
<td>Added parameter to hide options window</td>
</tr>
<tr>
<td>PointRight for Mac/Windows</td>
<td>“PointRight” name and logo used in various places</td>
<td>“PointRight” text changed to “TeamSpace”</td>
</tr>
<tr>
<td>MultiBrowse Droplet</td>
<td>Additional droplet window made UI cluttered</td>
<td>Functionality merged into TeamSpace GUI</td>
</tr>
</tbody>
</table>

Each of these changes reflects a limitation of incremental integration in a zero-admin environment. However, in all cases, the environment would still work without the code changes; it would just be less well-integrated with respect to boundaries and interface.

3.4.4 Extensibility

Until now we have focused on the TeamSpace usage scenario of PointRight and MultiBrowse. However, we emphasize that iROS Manager is a highly generic architecture, which would make it easy to extend TeamSpace further. For example, iStuff [3] and Patch Panel [2] could allow users to configure wireless buttons to perform different tasks. Workspace Navigator [7] could be used to log users’ documents and actions for them to revisit later.
A non-iROS application such as VNC could also be added to the TeamSpace installation to allow users to mirror their screens on the public displays. Since iROS Manager allows any application to be launched on demand, locally or remotely, a “Mirror” button could be added to the TeamSpace GUI to launch the VNC server on the local laptop and the VNC client on the remote display.

3.5 Lessons

3.5.1 Mundane Requirements

There are a few other requirements associated with a public computing resource that, while not represented in our zero-admin criteria, are nonetheless important. Some of these issues are as follows:

1. **File permissions:** The TeamSpace applications require read and write permissions to its own directory so that it can modify configuration files, and execute permissions to relevant applications. Beyond this, it needs no admin-level access, so it is amenable to a public system.

2. **System crashes:** Currently, if one of our public machines experiences a blue screen of death, there is no way for TeamSpace to elegantly handle the situation. In this case, our last line of defense is making public machines’ hardware reset buttons accessible to users. If a user resets a public machine, it will reboot into a clean state.

   In a more complex system, the machines could be monitored by another machine and automatically powered off in the event of a hard crash. However, for a system such as TeamSpace, the reset button approach is simpler and almost as effective.

3. **Machine ghosting:** In any public computing space, it is useful to save a state that is known to work correctly and ghosting machines periodically.

4. **Disk space:** If a system runs out of available disk space, processes may start to malfunction unpredictably. Wiping saved files at the end of a session partially solves this problem. Experience teaches us that whenever logging applications are used, special provisions must be in place to manage the large quantities of data created.
5. **Physical plant issues:** The intended uses of a workspace must be considered when deciding on a location. For example, early studies have indicated that TeamSpace users create a level of noise that distracts other library users.

Certain hardware configurations are prone to theft. For example, it is difficult to physically secure wireless or easily detachable input devices. In this regard, TeamSpace is advantageous because the only input devices required are the users’ own laptops.

6. **Privacy:** After each session, TeamSpace could destroy any files that were sent to the public machines via MultiBrowse. This would protect users who casually leave personal files on a public computer. This feature is planned for the future; in the mean time, TeamSpace espouses the same guideline used in other computer clusters at Stanford—private files should not be left in public folders.

7. **Security:** The Event Heap is built to be transparent and easy to snoop, which is an advantage for incremental integration and intermediation. However, users of TeamSpace are likely to type in passwords, including their university password, over PointRight. This poses a security risk. To address this issue, we are planning to switch TeamSpace over from Event Heap to the SSL-based Secure Event Heap in the near future.

### 3.5.2 Beyond Simple Restarts

As noted in Sect. 2.1, it is not always possible for applications to be fully restartable. When restarting alone will not restore availability to an application, a zero-admin architecture should find another way to restore availability or, if this is not possible, respond to the situation elegantly.

**Not All Restarts Result in a Clean Slate**

Sometimes an application is generally restartable, but if it it crashes in a certain way it can leave data in an inconsistent state.

One example comes from the Workspace Navigator system [7], which logs video frames, screen captures, and annotations from sessions in an interactive workspace such as the iRoom. One part of Workspace Navigator is a Capture application, which saves a frame from a video camera every 30 seconds, and then sends out archived images in response to
queries. If Capture is left unattended for several days, it fills up the available disk space, and then receives a disk error. The application still responds to queries, but if it is asked for a recent image, it will respond that the image is not available.

The situation can be remedied using a custom monitor, \texttt{CaptureMonitor}. This monitor polls the Capture application every five minutes requesting an image from one minute earlier. If the application responds that the image is not available, the monitor restarts the application. Then, if images are still not being captured, the monitor gives the application a “clean slate” by freeing disk space and restarting again.

Note that this monitor was written with a polling interval of five minutes, in contrast to the three-second interval of the Event Heap ping. This slower polling is desirable because each time the capture application is queried, it must send back an image file over the network. To save bandwidth, and because the capture service works on a fairly granular 30-second schedule anyway, the slower polling interval is desirable.

**Returning configuration files to a “clean slate”**

As another example, software that is not fully restartable may leave some configuration file or preference file in an inconsistent state when it crashes. Even if the data contained in this file is unimportant—for example, a list of recently opened documents—it may cause the application to misbehave if it becomes corrupted. This problem can also be fixed with the use of monitors: if an application is not responding, the monitor can simply attempt to restart it. If it is still not responding, the monitor can wipe all configuration files and try again.

**3.5.3 Implications for Zero-Admin Workspaces**

Our examination of TeamSpace has produced several lessons that apply to zero-admin workspaces in general.

First of all, any zero-admin architecture must support controlling arbitrary applications, not just a fixed set of applications, since the line between infrastructure and user-level applications is blurred in an incrementally deployed environment.

Second, while iROS Manager only implements the most basic type of restartability, this is adequate in many situations. With use of custom monitors iROS Manager can respond effectively to a wide variety of situations. It can detect malfunctions and restore an application-specific “clean slate.” Thus, restartability can be useful in maximizing uptime,
provided that it is not done overzealously in cases where restarting is too costly. In fact, the greater problem with restartability is the burden it places on developers, not the situations in which it fails.

In fact, much of the zero-admin ideal can be achieved simply by exploiting natural similarities between all applications, across multiple platforms. For example, iROS Manager achieves recoverability via restartability, which exploits the fact that all applications can be started and killed. Similarly, iROS Manager uses command-line arguments and environment variables to set application parameters.

However, as always, there is no “free lunch.” To achieve the aforementioned levels of recoverability and configurability in a zero-admin environment, the relevant applications must be crash-only and must be fully configurable using command-line arguments and environment variables. Likewise, it is not always possible for an incrementally integrated environment to abide by the Boundary Principle, because applications may wish to establish their own boundaries (see Sect. 3.3.4).
Chapter 4

Future Work and Conclusions

Outside the field of ubicomp, considerable research has studied ways to use restartability to maximum effect. Much of this work could be applied to iROS Manager to make it a more industrial-strength system for maximizing uptime. For example, duplicate instances of software components could be launched to provide redundancy. Memory and disk activity could be monitored in order to detect more situations in which an application is likely to be misbehaving. Also, the configuration files of applications could include a list of all dependencies between applications, both within a machine and between different machines. With this information, iROS Manager could recursively restart components and groups of components more intelligently.

As ubicomp comes from the research lab and into the real world, a key challenge will be balancing ease of administration with incremental integration. Our experiences with TeamSpace clarifies this issue and suggests possible solutions.

TeamSpace approaches the ideal of a zero-admin ubicomp environment while still being integrated from a set of heterogeneous software applications. It appears as a unified user experience, despite being composed of smaller building blocks. Importantly, the model of TeamSpace and iROS Manager is flexible enough so that it applies more generally to ubicomp environments in which the developers relinquish administrative control.

As we have shown, the goal of incremental integration is sometimes at odds with the ideal of zero-admin: it is impossible to automate administration of applications without imposing certain restrictions on those applications. iROS Manager illustrates that much can be accomplished in the direction of zero-admin while increasing the effort of integration only slightly.
As ubiquitous computing environments move from the research lab into the real world, we expect that different developers will differently emphasize zero-admin or incremental integration. Finding the right balance between the two will be a key issue in determining which architecture will underlie real-world ubicomp environments.
Acknowledgements

This research was performed under the guidance of Armando Fox. TeamSpace is a collaboration with Jeff Raymakers, Clara Shih, and Maria Croné, supported by Armando Fox, Terry Winograd, David Futey, and Richard Holeton. iROS Manager is based on work by Ulf Ochsenfahrt.
Bibliography


