Plethora: A Framework for Converting Generic Applications to Run in a Ubiquitous Environment

Zahid Anwar[†] Jalal Al-Muhtadi[†] William Yurcik[‡] Roy H. Campbell[†]
[†]Department of Computer Science
[‡]National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign (UIUC)
{anwar, almuhtad}@ uiuc.edu byurcik@ncsa.uiuc.edu rhc@cs.uiuc.edu

Abstract

Applications designed for ubiquitous computing environments need to be coded in a specific way in order to fully realize the benefits of ubiquitous computing. Currently, applications for ubiquitous computing environments either need to be rewritten entirely to benefit from ubiquity, or special wrappers need to be written and customized for particular applications to provide limited compatibility. We argue that the real-world deployment of ubiquitous computing will be realized when users can migrate and use the applications they are familiar with in their daily lives with minimal effort. Furthermore, these applications should automatically benefit from typical ubiquitous computing features including multi-device support, run-time adaptation, environment-independence and context-awareness. In this paper we present a framework that allows us to port any generic application to the domain of ubiquitous computing without having to rewrite the code from scratch. We have experimented with the framework in our prototype ubiquitous computing platform known as Active Spaces. This has allowed us to explosively increase the number of applications supported by our Active Space.

1. Introduction

When Mark Weiser coined the phrase "ubiquitous computing" in 1988 he envisioned computers embedded in walls, tabletops, and everyday objects. In ubiquitous computing, a person may interact with hundreds of computers at a time, each invisibly embedded in the environment and wirelessly communicating with each other. This dream has failed to become a reality almost two decades later because there is no easy way to design applications for distributed environments. Ubiquitous computing environments should typically contain a large number of applications complementing each other's utility. For example, a meeting may require a presentation application for demonstration, a word processor application for taking minutes, and a paintbrush application for notes.

For example, our research environment at UIUC consists of an active space that executes applications across a variety of computers and display monitors. Our software infrastructure also consists of middleware services that allow applications to be instantiated on, and moved between, different machines in this environment. For example, a drawing made by the meeting chair on his desktop can be viewed at the same time on other desktops being used by the various attendees.

Although we have a wide range of applications designed and running in our active space, whenever the need for a new application arises it is painful to rewrite anew according to our programming model. Currently, applications for ubiquitous computing environments either need to be: (1) rewritten entirely to benefit from ubiquity or (2) special wrappers need to be written and customized for particular applications to provide limited compatibility. Often these wrappers require applications to provide an SDK or COM object that developers can use to customize the application for their purpose. In either scenario, application-specific code needs to be written. The Plethora framework allows us to execute many generic applications in our active space without having to be rewritten. Plethora also eliminates the need for source code analysis since it works with binaries. It allows the use of complicated and huge applications in our active space regardless of whether they support the COM standard or not.

In this paper we present Plethora –a framework built on top a Gaia, a meta-operating system that brings the functionality of an operating system to physical spaces. Plethora allows an ordinary single user, single machine application to be used in a distributed fashion with controllers and views executing on different machines enabling multiple users to interact with them collaboratively. In addition the framework is programmable and enables users to program "follow-me" functionality into generic applications. It also adds the concept of user sessions for these applications so that a user can save and restore the entire state of all his applications that are running in the room at any point in time. Using Plethora a user can program an active space to actuate his applications based on certain triggers and events. For instance he can program a lights-controller application to dim the lights of the room whenever he leaves provided there is no one working there.

The rest of the paper is organized as follows: Section 2 describes some of the background and related work required. Section 3 describes the design and implementation of Plethora and Section 4 describes Plethora applications in our Active Space as well has how their lifecycle is managed. We evaluate and compare our work to related research and discuss some of the issues we encountered in Section 5. Finally, we touch on some future work and conclude the paper in Section 7.

2. Background

In this section, we define what we mean by ubiquitous and what features an application should have to qualify as one.

2.1. Ubiquitous Applications – A Definition

Our working definition of ubiquitous computing is:

- computing invisible to users (it disappears)
- computing independent of physical location (ubiquitous)
- computing independent of display platform (independent of GUI size, shape, I/O bandwidth, and resolution) [6]
- computing that is not tied to instances of time (asynchronous)

In our active space environment we have an application known as *Active Presentation*, which can be conceptualized as the ubiquitous version of a PowerPoint type multimedia presentation application. The application exports functionality to present slides in multiple displays simultaneously, supports moving and duplicating slides to different displays during the presentation, and allows moving and duplicating the input sensor that controls the presentation to different devices. The presentation manager is based on the Gaia application framework and uses PowerPoint to manipulate the slides (using the COM interface). We will use this application as an example to illustrate the following features ubiquitous applications must support:

Multi-Device Support. With a variety and number of I/O devices to choose from, the user should be free to control his slides from the buttons on his smart watch, PDA, cell phone to the windows dialog on a regular desktop. Similarly the slides can be displayed on the overhead projector as well as the conference attendees' laptop computers simultaneously.

Session Maintenance. With a host of devices and resources tied up by a single user application there is a need for managing the association of applications with users. We define a user session as a set of applications and files that a user interacts with [4]. A user session also includes state information and customization options selected by the user. If a user moves out of the vicinity of the display he is using, the application will automatically suspend the session. When a user is detected in the vicinity of another display or workstation, the session is automatically migrated and resumed at that display or workstation. In effect, users can resume their work anywhere and anytime without having to remember to save the latest changes or to worry about copying their data to a removable disk. This allows ubiquitous applications to become environment-independent.

Location Awareness. The whole idea of ubiquitous applications following users around falls apart if the environment has no way of determining where the user is currently located. For instance, in our active space environment, applications can keep track of the user via an indoor location service that can have several location-sensing technologies such as radio frequency ID (RFI) badge detectors, Bluetooth, WiFi base stations and Ubisense [16].

Contextual Information. Context-awareness is a key issue in ubiquitous computing. Applications should be able to respond and adapt to changes in the environment. Furthermore, applications can be categorized into various contexts. For example, our Active Space presentation may belong to a 'meeting' context whereas a Music Player application may belong to a 'light-entertainment' context. Since these two contexts may conflict, the Active Space scheduling policy should be carefully designed. For instance, a spontaneous game should not necessarily block a planned conference meeting if the users of the two applications happen to run in the same time slot.

2.2. Gaia Active Spaces Prototype

The computational infrastructure of our ubiquitous computing environments is controlled by the Gaia OS [14], a distributed meta-operating system that runs on top of existing computer operating systems. The Gaia OS Kernel provides a collection of services that orchestrate the many heterogeneous devices and services present in the environment to enable application development. It integrates physical spaces and their ubiquitous computing devices into a programmable computing and communication system. Gaia provides the infrastructure and core services necessary for constructing ubiquitous computing environments. Component-based applications developed for Gaia OS use an application framework [Roman03] inspired by the Model-View-Controller design pattern (see Figure 1). We refer to this framework as Model-Presentation-Controller-Coordinator (MPCC).

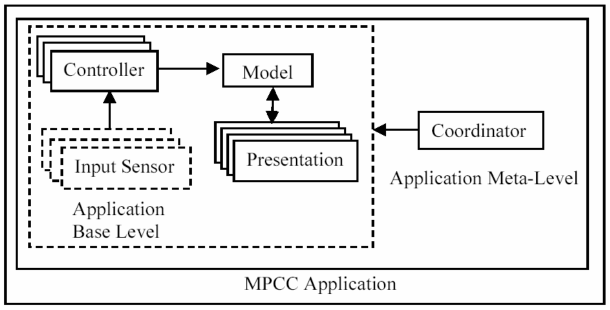


Figure 1. The Gaia MPCC Model

2.3. Application Framework

The MPCC framework separates applications into five different components: (1) a model to implement application logic and store the application state, (2) one or more presentations to provide an output mechanism for the model, (3) one or more controllers to provide input to the model, (4) an adapter to translate controller requests into method calls on the model's interface, and (5) a coordinator to manage the application composition and allow dynamic binding of application components. The application model uses events to notify presentations and controllers about changes in the application's state. As a result, presentations and controllers may invoke methods on the model's interface to obtain the new application state or to trigger new changes. In Gaia, each component is implemented as a CORBA object [5].

Although straightforward in design, the MPCC framework can be difficult to implement. The Model and Viewer are typically closely coupled, sharing global variables and pointer references, such that decoupling by placing them in separate address spaces with only CORBA interfaces for communication represents a significant programming challenge.

3. 'Ubiquitizing' Applications

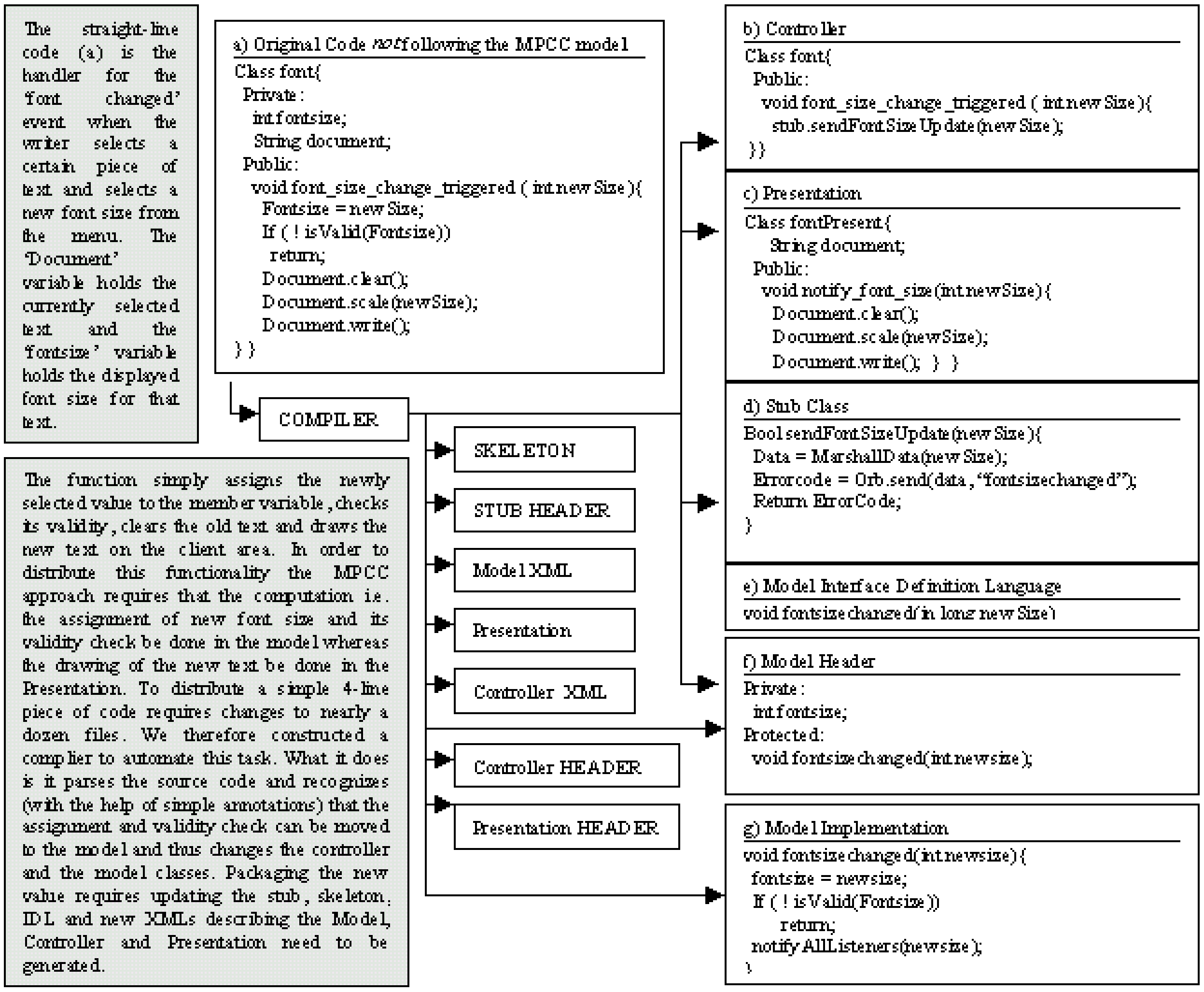
3.1. The Need

People need control of information in many different formats – data, text, graphics, video, audio, voice-mail, Email, fax, bookmarks, documents, personal organizers - and the amount of data is constantly growing. Because there is little opportunity to distribute this information, it becomes increasingly inaccessible – objects either have to be endlessly copied or risk becoming accessible only from one computer.

Significant research is going on to develop a 'killer' application. We propose that ubiquitous applications do not need to be 'special'; instead generic applications can be used in a ubiquitous environment without rewriting them. In this paper we describe a number of popular applications that have been given the properties of ubiquitous computing using our framework. They have been made location-aware in the sense that they automatically detect when their user is moving to another machine. They are collaborative and allow multiple users to work on them jointly by distributing their displays and input sensors across different machines. They can be bridged with other types of applications to enhance their functionality. We also show how an operating system can be programmed according to user preference.

3.2. A Compiler-Based Approach

We consider more complex classes, for instance word processing. This class of application does not easily lend itself to being subdivided into the traditional distributed model-view-controller architecture that we have previously described. This is due to the blurred boundary between the view and the controller that makes it difficult to differentiate between the input sensor and presentations. Additionally there are more features that word processors expose. A piece of typewritten text can be made bold, italic, underlined, resized, bulleted, etc. The MPCC design requires an event be sent to the model to update the variable there, for each such 'font change' action. More precisely, a function has to be added to the CORBA stub for each trigger, the values marshaled and sent over to the model to have the appropriate interface exported. After the computation is performed, the result is returned when timing and consistency issues have been taken into account. One way to automate this task, which was also our first approach, was to build a compiler that would take a straight-line piece of code and generate the appropriate stubs, skeletons, marshaling and demarshaling code (See Figure 2).


The straight-line code (a) is the handler for the 'font changed' event when the writer selects a certain piece of text and selects a new font size from the menu. The 'Document' variable holds the currently selected text and the 'fontsize' variable holds the displayed font size for that text.

a) Original Code *not* following the MPCC model

```
Class font{
 Private :
  int fontsize;
  String document;
 Public:
  void font_size_change_triggered ( int new Size ){
   Fontsize = new Size;
   If ( ! isValid(Fontsize))
    return;
   Document.clear();
   Document.scale(newSize);
   Document.write();
} }
```

The function simply assigns the newly selected value to the member variable, checks its validity, clears the old text and draws the new text on the client area. In order to distribute this functionality the MPCC approach requires that the computation i.e. the assignment of new font size and its validity check be done in the model whereas the drawing of the new text be done in the Presentation. To distribute a simple 4-line piece of code requires changes to nearly a dozen files. We therefore constructed a compiler to automate this task. What it does is it parses the source code and recognizes (with the help of simple annotations) that the assignment and validity check can be moved to the model and thus changes the controller and the model classes. Packaging the new value requires updating the stub, skeleton, IDL and new XMLs describing the Model, Controller and Presentation need to be generated.

b) Controller

```
Class font{
 Public:
  void font_size_change_triggered ( int new Size ){
   stub.sendFontSizeUpdate(new Size);
 }}
```

c) Presentation

```
Class fontPresent{
   String document;
 Public:
  void notify_font_size(int newSize){
   Document.clear();
   Document.scale(newSize);
   Document.write();  }  }
```

d) Stub Class

```
Bool sendFontSizeUpdate(new Size){
 Data = MarshallData(new Size);
 Errorcode = Orb.send(data, "fontsizechanged");
 Return ErrorCode;
}
```

e) Model Interface Definition Language

```
void fontsizechanged(in long new Size)
```

f) Model Header

```
Private :
 int fontsize;
Protected:
 void fontsizechanged(int newsize);
```

g) Model Implementation

```
void fontsizechanged(int newsize){
 fontsize = newsize;
 If ( ! isValid(Fontsize))
    return;
 notifyAllListeners(newsize);
}
```

Figure 2. A Compiler for 'Ubiquitizing' Code

The problem with this approach is how to decide which functionality to migrate to the model, and which to keep behind. On one hand we would like a dumb viewer and have all the computation in the model, however, we also do not want to send it. For instance, negative font values would have to be rejected. Thus the compiler has to make a tradeoff between network latency and client complexity. Similarly, if the user vacillates, should the compiler allow a user to make an ultimate decision before propagating it to the model or take a more reactive approach? Although this approach is elegant, it still requires effort on the user's part. While writing each line of code, the developer may have to add annotations to signal the compiler which code he doesn't want moved. Aside from this, the problem of rewriting the entire application still remains.

We decided to take a different approach, is it really necessary to have a centralized approach (decoupling the model from the viewer and controller)? We propose a decentralized approach where each application has its own model but the behavior of different models is synchronized so the final output from all viewers remains the same.

3.3. The Plethora Framework

Our approach is based on the observation that model behavior is usually predictable. The algorithms will give a deterministic output to a particular input permutation provided all the various parameters and variables on which the model is dependent are kept constant. For example, a calculator application will always return the same result for 22 / 7, a word application will reformat the text in the same way every time someone presses the 'Bold' button, and a music player application will play the same melody when a song is selected. There are some exceptions to this rule namely applications that:

1. adapt themselves to the capabilities of the machine such as size of memory, display and occasionally performance
2. utilize machine learning or genetic algorithms
3. actively use randomization based on some pseudo-random number generators
4. have dependencies on other active system processes

The third point, the use of randomization, is usually a characteristic of gaming applications, for instance a Windows Minesweeper game will randomly generate a new set of hidden mines every time its instantiated, to make the game more challenging. The fourth point relates to the class of applications known as services. For example of such services we point to anti-virus software that augment other applications and operating system functions that are not directly used by the user. We do not include these classes of applications to those that can be 'ubiquitized,' since it is not clear how distributed versions of these applications would benefit a user.

The first two classes of applications need to be addressed because they are by far the most common. For instance a word processor may automatically choose a default font if it discovers that the one the user selected is not installed or available to the operating system. For example, a painting application may resize its canvas region depending on the display size and resolution supported. A word processor may also display limited learning capabilities by remembering what favorite formatting options the user normally uses and modifies the template to facilitate the user henceforth.

Figure 3 shows how we surround our applications with a 'virtual machine' environment, a technique similar to process migration check pointing schemes used for intercepting API calls [1,15]. API interception is a technique used to acquire crucial information about the process like which files are opened, which graphics devices are used, or which network connections are established - providing the application with new facilities without changing the application's code. However, the artificial environment we create for the application serves the exact converse purpose - namely controlling the resources available to the application so that its behavior is more predictable.

3.4. Plethora Implementation

The current implementation of Plethora targets MS Windows applications. We believe that the same principals can be applied to other platforms especially Linux. The VM takes the responsibility of creating, managing and terminating the application.

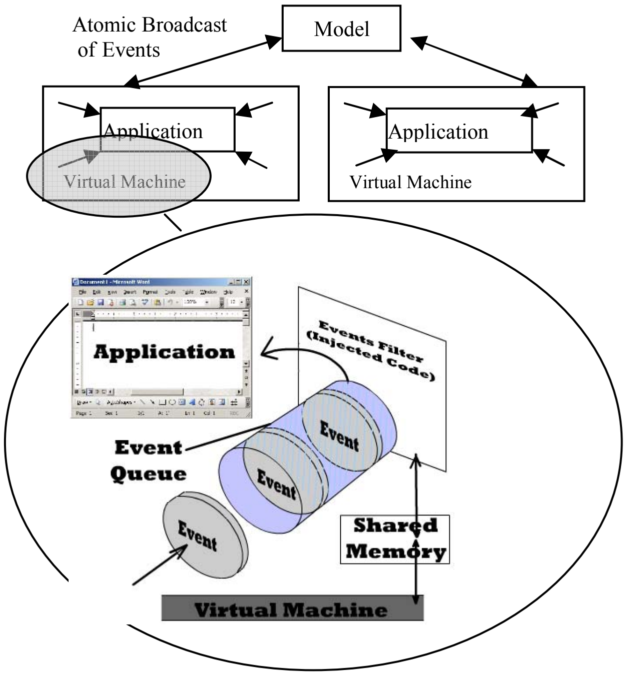


Figure 3. Plethora Virtual Machine Architecture

We use a CORBA-based scripting language to create our Gaia Applications and use the Windows 'CreateProcess' API to launch an application after searching the registry for the executable name and path. Any external dependencies the application has along with libraries are version controlled and preset to values that are predetermined across all the machines in the space. After the application has been successfully launched the VM adds system 'hooks' to the application, to control all message traffic sent to or from it. Windows Hooks are one of the few ways to inject your code into a remote process's address space. We require that whenever the controlled application examines its message queue using a 'GetMessage' or 'PeekMessage' function then the VM be informed about it first. Therefore our spy code was written in a DLL and used 'SetWindowsHookEx' to map our DLL to the application. Our injected code subsequently communicates the events to the VM using the 'shared memory' IPC mechanism.

The next step, after injecting hooks, is to disable the various functions Windows provides to directly change the appearance of the application. Options such as minimize, maximize, terminate, dock, cascade, resize directly, are all disabled so that any attempt the user makes to do so are captured by and executed through the VM. The VM then queries the model to check if the application has any state associated with it, if so then it retrieves the state and applies that to the captured application. Table 1 shows three scenarios where an application state present in the model needs to be reapplied.

Table 1: Scenarios for Storing State Information

Scenarios where application state needs to be reapplied.	**Examples**
(1) If an application is already running in the space and the user decides to add another controller.	A user working on a particular document and his friend just walks in to help him with it, i.e. collaborate on it.
(2) The controller is migrated/ duplicated from one machine to the other.	A user moves to another room and wants his application to follow him.
(3) A user session is being retrieved from scratch. A session is defined as the user environment the system saves when a user leaves a room.	A user opens his office early morning and the system starts up his favorite Email application and newsreader.

3.5. Capturing Application State

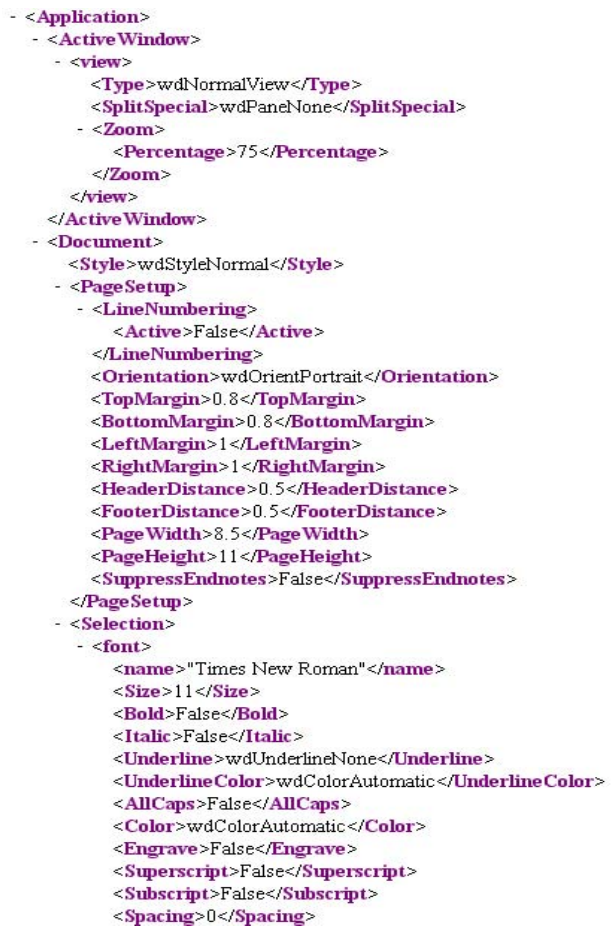
An application process may consist of data regions that include statically and dynamically allocated data. For example, to complete the memory components of the process state may require calls to malloc or new, program stack and the value of registers, stack pointer and program counter. A process may have open files and inter-process communication channels with data in transit.

An application state is determined by a group of mechanisms. We initially made an unsuccessful attempt at implementing a process memory snapshot similar to what the Windows 'Hibernate/Suspend' option does at the Operating System level. Since we were primarily working with the Windows Operating System, we did not have the privilege to take a snapshot of the process memory (stacks and registers) since they are not visible to the user. It was also more difficult because of the need to save the state of the dependency processes.

Next we tried logging all the different events that had occurred since the application had been started until time to save state. Although such a recording of events is not the *true* state of the application, the state can still be reconstructed by replaying those events on to a new instance of the application. It should be noted that this could lead to a redundancy. For example - if a user initially signs his name to a document a dozen times and then later deletes it every time (because he does not like the font or style) and then ultimately decides not to have a document signature after all – this would lead to the significant recording of redundant events.

We ultimately adopted a partial approach as shown in Figure 4 for the case of Microsoft Word; the developer is given freedom to specify what constitutes the state of an application using an XML format. This is in case the partial state information the VM saves is not sufficient (likely to be a rare scenario). By default the VM saves the contents of all *child windows* in the application. It does this by a call to the 'Enumwindows' and copying the displayed text/data/image for each of the enumerated windows.

We 'ubiquitized' a group of windows applications and the technique worked very smoothly for each application. For instance, with Microsoft Word the state stored was the document typed in the client area together with the formatting, current cursor position, and button values on the tool bars.



```
- <Application>
  - <ActiveWindow>
    - <view>
        <Type>wdNormalView</Type>
        <SplitSpecial>wdPaneNone</SplitSpecial>
      - <Zoom>
          <Percentage>75</Percentage>
        </Zoom>
      </view>
    </ActiveWindow>
  - <Document>
      <Style>wdStyleNormal</Style>
    - <PageSetup>
      - <LineNumbering>
          <Active>False</Active>
        </LineNumbering>
        <Orientation>wdOrientPortrait</Orientation>
        <TopMargin>0.8</TopMargin>
        <BottomMargin>0.8</BottomMargin>
        <LeftMargin>1</LeftMargin>
        <RightMargin>1</RightMargin>
        <HeaderDistance>0.5</HeaderDistance>
        <FooterDistance>0.5</FooterDistance>
        <PageWidth>8.5</PageWidth>
        <PageHeight>11</PageHeight>
        <SuppressEndnotes>False</SuppressEndnotes>
      </PageSetup>
    - <Selection>
      - <font>
          <name>"Times New Roman"</name>
          <Size>11</Size>
          <Bold>False</Bold>
          <Italic>False</Italic>
          <Underline>wdUnderlineNone</Underline>
          <UnderlineColor>wdColorAutomatic</UnderlineColor>
          <AllCaps>False</AllCaps>
          <Color>wdColorAutomatic</Color>
          <Engrave>False</Engrave>
          <Superscript>False</Superscript>
          <Subscript>False</Subscript>
          <Spacing>0</Spacing>
```

Figure 4. Application State Stored in XML

Other implicit states such as whether the document is in 'Print Layout' or 'Normal' view or what line number the user was currently typing are ignored. The developer may choose to add these as part of his state from a graphical user interface or type them in an XML document. The Graphical User interface provided lists the COM interfaces that the application exports which the user can click on to show that he is interested in saving the data value it returns.

3.6. Management of Application Life Cycle

The natural question that arises from this is how to manage all the applications since we have disabled direct control. The answer is that the scheduler for the active space meta-operating system Gaia handles them for the user with the help of a *context engine* [12] and a *space repository viewer* as shown in Figure 5. The latter is a store of the entities registered in the system, such as different devices, displays and users, whereas the former keeps track of activities going taking place in the space for instance meetings, brainstorming sessions, free relaxation time, and classroom lectures. The *Location Service* is the source of location information for all location-sensitive applications [11]. It fuses data from multiple sensors, resolves conflicts, answers object-based/region-based queries based on subscriptions for location-based conditions, and notifies applications when conditions become true. In order to subscribe to the service, one has to create spatial regions and associate different kinds of properties with these regions.

In addition, we also confronted the problem of actuating third-party applications that were not designed to execute in a *Smart Room* environment. To handle such cases, Plethora requires a user to fill out a GUI-based form (a smaller version of which is shown in Table 2). The VM uses this information to verify whether applications can indeed run on the devices the user has specified. This avoids situations where the user leaves his an application running in his Smart Office and Plethora attempts to move a document to his PDA but instead crashes the PDA since there is not enough memory in the PDA to support an application such as Microsoft Windows.

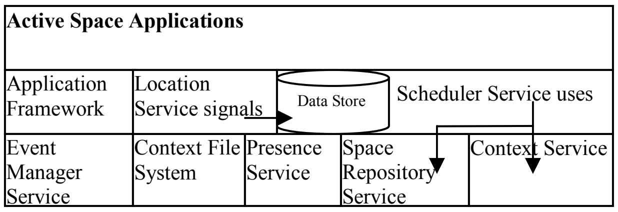


Figure 5: The Gaia Scheduler Service

Table 2: Sample User Form for Categorizing Generic Applications for 'Ubiquitization'

Application	**Properties**																			
	Output								**Input**						**Processing Resources**					
	Display				Sound				Keyboard			Mouse								
	None Needed	Optional	Required		Mute	Optional	Required		None Needed	Optional	Required	None Needed	Optional	Required	Smart Watch	PDA	iPAQ	Cell Phone	Laptop	Desktop
			Flexible	Fixed Size			Volume Flexible	Volume Fixed												
Microsoft Word			●		●						●		●						●	●
Microsoft Excel			●		●						●		●						●	●
MP3 Player		●					●			●			●			●	●	●	●	●
Microsoft Paint			●		●				●					●					●	●
PPTPresentation				●		●				●			●		●	●	●		●	●
Email Reader			●		●					●			●			●	●	●	●	●
Media Player			●				●		●				●					●	●	●

3.7. Location–Based Actuation

The scheduler service can be configured to perform certain actions based on event triggers. We define an *action* to be the execution of a method call on an *entity* in a particular *space*. For example, ACTION1 may be defined as *start* in the application *Microsoft Word* in the *Active Space Prototype Lab*. Actions can have properties such as delayed/partial execution and can be rolled back or cascaded. Actions can be unconditionally executed but typically they are executed as part of a condition. The conditions on which actions are triggered are called *events*. Event conditions are considered commutative and associative so more than one conditional event can be joined by logical connectives.

Event properties include *location*, *time,* and *targets*. For example, EVENT1 can be when *user* approaches LOCATION1 **and** time is between 9 AM to 5 PM.

Location applies to *entities* having proximity to *targets*. Entities are objects in *space* that have a location sensor. *Target* is an area of interest identified by its Cartesian coordinates. Entities can be *inside, outside*, *near* a target, or they could *enter* or *exit* it. For example, LOCATION1 can be defined as a circle of radius 2m at the center of an active space with coordinates 0,0,0. The composition of *actions*, *events*, *spaces*, *entities*, and *users* is termed as a *behavior*. Behaviors have a textual description and can be categorized into *domains* as shown in Figure 6.

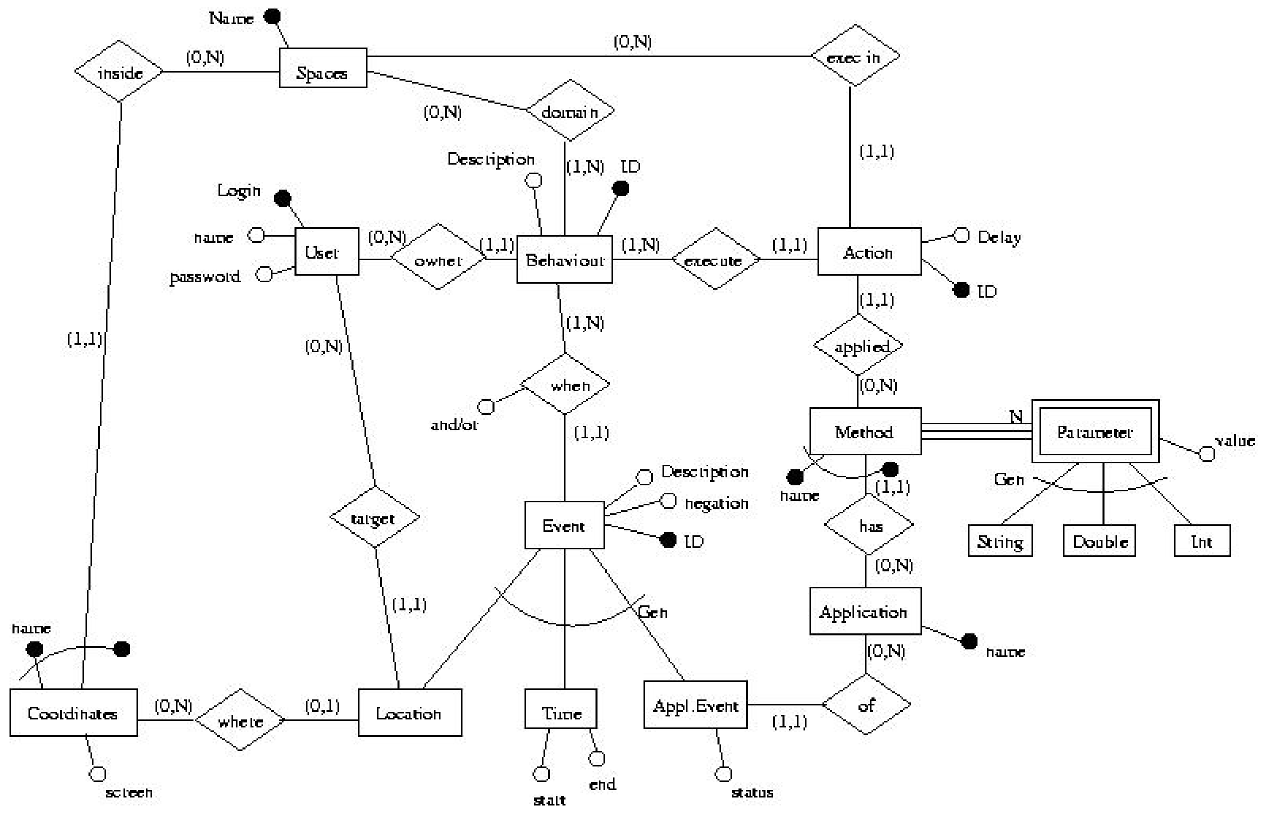


Figure 6. Studying the Relationships Between Entities, Actions, Events, Spaces, and Users

4. Related Work

There are many research projects working to create an infrastructure to support ubiquitous computing. Many of these efforts propose new programming models and try to support the unique nature of pervasive applications and devices by providing systems support from the ground-up. As a result these research projects support only a small variety of applications. What differentiates our research is that we provide support for 'ubiquitizing' popular existing applications via a platform-independent middleware that can be layered on top of any operating system.

Microsoft's Remote Desktop Connection (RDC) allows users to connect to a terminal server or another computer running Windows. The advertisement for RDC states: "wherever you are, if you have Internet access, you can work as if you were sitting at your home computer". RDC is extended into the realm of pervasive computing by running a single server and clients on any machine a user may wish to display upon. This simplistic approach has three primary drawbacks. First, while RDC provides a current desktop picture snapshot to all clients *and all the computation occurs centrally.* Second, RDC has no concept of *state*. Third, as Figure 7 shows, network bandwidth can be more than three times that of Plethora since RDC broadcasts the entire desktop constantly while Plethora only transmits events as they occur.

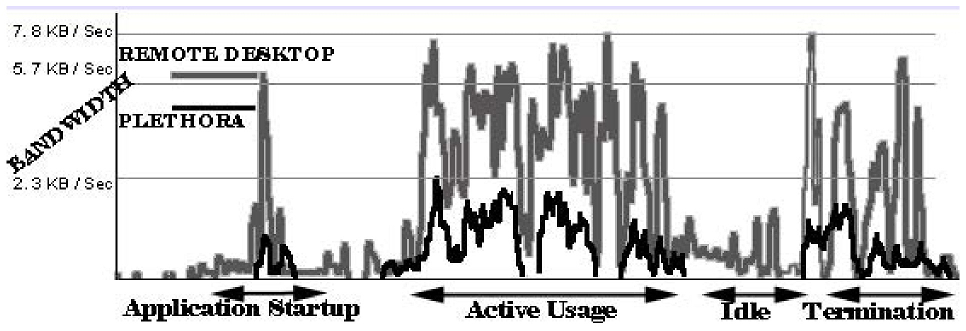


Figure 7. RDC vs. Plethora in Terms of Network Bandwidth for an MSWord Application. Note that the test period consists of multiple phases: (1) application startup, (2) active usage, (3) idle, and (4) termination. RDC showed significantly more bandwidth consumption even during the idle phase.

A descendent of Microsoft's *Digital Dashboard* [9] is the *SharePoint Portal Server* is designed for users to share documents for review and manage meetings using a web services based version control system. Plethora allows collaboration of existing applications without actually requiring the setting up of a centralized server and portal services with user profiles and web interaction.

MIT's Project Oxygen, takes an AI goal-oriented approach to ubiquitous computing [10]. The goal-oriented approach centers around three concepts: (1) goals denoting intents, (2) techniques that satisfy intents, and (3) a *Planner* that matches goals and techniques. The *Planner* is the heart of the goal-oriented system, searching through a plan tree of goals and techniques. Our work is similar in the sense that we are enabling end-user control of the ubiquitous computing environment. However, instead of allowing users direct control of an application and focusing on HCI issues surrounding this task, we also provide a general framework that supports user control of application functionality. Software engineering is critical in this project since Oxygen is the backbone for implementations that the Planner creates whereas Plethora has no such restriction.

Rhapsody [13] is a UML based application development environment that allows a user to specify Use Cases and their corresponding behavior as sequence, collaboration, state charts, and activity diagrams. Rhapsody also provides tools to convert Use Cases into executable models. This design methodology does not allow complete code reuse and the design framework for ubiquitous applications is completely user specific which is not scalable.

One.World's [3, 7] four foundation services – (1) a virtual machine, (2) tuples, (3) asynchronous events, and (4) environments - provide basic building blocks for creating adaptable applications. However, since they have built the kernel and its services from the ground up, applications have to be completely rewritten. In fact they have recruited outside developers to rewrite the 'electronic laboratory assistant' for One.World.

Stanford's Interactive Workspace project uses an *Event Heap* to allow application interaction with decoupled applications [8]. The Event Heap is derived from a tuplespace. Events within the Event Heap are typed, self-describing, tuples stored in a centralized location. Applications can post events and remove them from the heap either destructively or nondestructively. Applications are designed by experts to handle and consume certain types of events that are meaningful to the particular application. Plethora differs in that we allow non-expert users to design applications.

5. Future Work

One stark observation a user makes when using Plethora is that event propagation is slow. Atomic ordering on all events makes displays update themselves with a delay equal to that of the slowest receiver. Therefore it is quite common for a user who is using a ubiquitous "Paint Brush" application to notice that the circle he drew on the primary display may appear half a second later on other displays. This was generally found to be a secondary concern to the users and can be fixed if a relationship equation can be determined between the various types of events. Events that are not related can then be grouped and transmitted together even if the timestamp of one is before the other.

We plan to extend Plethora for frameworks such as Linux and envision that it should be an easier exercise since the event-capturing module of the virtual machine can be designed as a kernel module giving us additional flexibility. Ultimately, this will allow us to support a more generic class of applications.

We are also working on a cross platform VM that will allow us to duplicate functionality. For example, we plan to support the Windows-specific applications like Microsoft Office on Linux by broadcasting events along with the subset of the image to the Linux-VM counterpart using image processing techniques. No doubt more support from operating system manufacturers and application developers to control process behavior would aid Plethora.

We are also planning to integrate with Gaia *Clicky* [2] to support moving generic applications across displays to simulate one integrated display wall.

6. Conclusion

In this paper we present Plethora - a framework for 'ubiquitizing' existing applications. It works by synchronizing application models across devices using the concepts of event filtering, code injection, and state duplication. We report here that Plethora works well for deterministic classes of applications and we look forward to reporting more results as we embark on the future work efforts we describe.

More than a decade after Mark Weiser came up with the concept of "Computing crawling out-of-the-woodwork", the idea of ubiquitous computing is still far from being realized. The primary reason for this is that research developers have concentrated on 'reinventing the wheel', finding the ultimate application, and developing separate ubiquitous systems from the ground up. The contribution of Plethora is that 'ubiquitizing' existing popular applications avoids these three problems and may be the catalyst needed for users to grasp the concept of ubiquitous computing as well as stimulating the development of new ubiquitous applications (that do not have to be rewritten).

7. References

[1] H. Abdel-Shafi, E Speight, and J.K. Bennett, "Efficient User-Level Thread Migration and Checkpointing on Windows NT Clusters," *USENIX Windows NT Symp.,* 1999.

[2] C.R. Andrews, G. Sampemane, A. Weiler and R. H. Campbell, "Clicky: User-Centric Input for Active Spaces." *University of Illinois AT Urbana-Champaign Dept. of CS Technical Report UIUCDCS-R-2004-2469,* 2004.

[3] L. Arnstein, R. Grimm, C.-Y. Hung, J. H. Kang, A. LaMarca, S. B. Sigurdsson, J. Su, and G. Borriello, "Systems Support for Ubiquitous Computing: A Case Study of Two Implementations of Labscape," *IEEE Intl. Conf. on Pervasive Computing and Communications (PerCom)*, 2002.

[4] D. Carvalho, R.H. Campbell, G. Belford, and D. Mickunas, "Definition of a User Environment in a Ubiquitous System," *International Symposium of Distributed Objects and Applications,* 2003.

[5]The Common Object Request Broker (CORBA): Architecture and Specification, Revision 2.2, July 1998. <URL:http://www.omg.org/corba/c2indx.htm>

[6] K. Gajos and D.S. Weld, "Automatically Generating User Interfaces for Ubiquitous Applications," *Intl. Conf. On Intelligent User Interfaces*, 2004.

[7] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. "Programming for Pervasive Computing Environments," *University of Washington Dept. of Computer Science Technical Report, UW-CSE-01-06-01*, 2001.

[8] B. Johanson and A. Fox, "The Event Heap: A Coordination Infrastructure for Interactive Workspaces," *IEEE Workshop on Mobile Comp. Systems and Applications*, 2002.

[9] Microsoft SharePoint <http://www.microsoft.com/sharepoint/evaluationoverview.asp>

[10] J. M. Paluska, "Automatic Implementation Generation of Pervasive Applications," *M.I.T Student Oxygen Workshop*, 2004.

[11] A. Ranganathan, J. Al-Muhtadi, S. Chetan, R.H. Campbell, and M. D. Mickunas, "MiddleWhere: A Middleware for Location Awareness in Ubiquitous Computing Applications," *IEEE Intl. Conf. on Pervasive Computing and Communications (PerCom),* 2004.

[12] A. Ranganathan, J. Al-Muhtadi, and R.H. Campbell, "Reasoning about Uncertain Contexts in Pervasive Computing Environments," *IEEE Intl. Conf. on Pervasive Computing and Communications (PerCom),* 2004.

[13] Rhapsody, "UML Application Development Platform for Pervasive Computing," *I-Logix Product Documentation*. <http://www.nohau.se/products/uml/>
[Roman03] M. Román, B. Ziebart, and R.H. Campbell. "Dynamic Application Composition: Customizing the Behavior of an Active Space," *IEEE Intl. Conf. on Pervasive Computing and Communications (PerCom),* 2003.

[14] M. Román, C. K. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, and K. Nahrstedt, "Gaia: A Middleware Infrastructure to Enable Active Spaces," *IEEE Intl. Conf. on Pervasive Computing and Communications (PerCom)*, 2002.

[15] J. Srouji, P. Schuster, M. Bach, and Y. Kuzmin, "A Transparent Checkpoint Facility On NT," *USENIX Windows NT Symposium,* 1998.

[16] *Local Position System and Sentient Computing*, Ubisense Webpage <http://www.ubisense.net/>