AN APPLICATION FRAMEWORK FOR ACTIVE SPACE APPLICATIONS

BY

MANUEL ROMAN

Ing., Ramon Llull University, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

Abstract

Ubiquitous computing lays the foundation for a future where people’s living and working spaces are interactive and programmable. Users interact with their offices, homes, cars, malls and airports to request information, benefit from the resources available, and configure the habitat’s behavior. Data and tasks are always accessible and are mapped dynamically to convenient resources present in the current location. Users may extend the habitat with personal devices that seamlessly integrate with the environment. Such user-oriented interactive environments may require a novel software infrastructure to operate their resources, sense context properties, and assist in the development and execution of applications.

I present in this thesis an application framework and the supporting infrastructure to assist in the development of user-centric, resource-aware, context-sensitive, multi-device and mobile applications. The framework consists of a distributed component-based infrastructure, a mechanism to define applications once and adapt them to different environments automatically, and a configurable management interface.

To my family, for their unconditional support throughout all these years.

Acknowledgments

This thesis would have never been possible without the help of several people. First of all, I would like to thank my advisor Roy H. Campbell for his support, his open minded attitude, and his creativity. He always gave me freedom to choose my own path, and helped me to get back on track when I got lost. I have learnt a great deal from him, especially the importance of a positive attitude to address new challenges. I am really grateful to him.

I would also like to thank Professors Nahrstedt, Mickunas, and Kravets for their support and for their advice when I asked them for help. I am thankful for having had the opportunity of working with them.

These four years would have never been possible without my friends at the SRG group. I would especially like to mention Fabio Kon, for his kindness, sense of humor and humanity. It was a pleasure to work with him during the last three years. I had also the pleasure of interacting with many other people: Anda, Prasad, Chris, Apu, Paola, Jalal, Renato, Cristina, Brian, Suvda, Dav, Samart, Rodrigo, Ping, Miguel, Luiz, and many other. They made my stay in Urbana-Champaign really comfortable.

I cannot forget about Elisabet, Josep Maria, and Miquel, who guided me and fostered my interest in research during my initial college years. Finally I would also like to thank Fulbright for giving me such a wonderful opportunity. The whole experience has helped me to obtain a PhD, but more importantly, has helped me to grow as a person and to discover a new country.

Table of Contents

List of Figures....xi

Introduction 1

1.1. Long Term Vision....3

1.2. Thesis Scope....4

Chapter 2 Gaia OS: An Active Space Meta-Operating System....6

2.1. Gaia Kernel....7

2.1.1. Event Manager 8

2.1.2. Context Service 8

2.1.3. Presence Service....10

2.1.4. Space Repository....11

2.1.5. Context File System 14

2.2. Application Framework 17

2.3. Gaia Management Tools....18

2.3.1. Lua: Gaia's Scripting Language....18

2.3.2. Bootstrap 19

2.4. Gaia OS and Traditional Operating Systems....20

2.5. Gaia OS Evolution 23

2.6. Gaia Evaluation....24

2.6.1. Platform and Language Portability 25

2.6.2. Application Portability and Extensibility 25

2.6.3. Recovery from Partial Failures 26

2.6.4. Scalability....28

2.6.5. Explicit Application Development Support 29

Chapter 3 Application Framework Overview....31

3.1 Application Framework Scope 33

3.2 Active Space Application Challenges 35

3.2.1 Resource-Awareness 35

3.2.2 Multi-Device 35

3.2.3 User-Centrism 36

3.2.4 Run-Time Adaptation 37

3.2.5 Mobility 38

3.2.6 Context-Sensitivity 39

3.2.7 Active Space Independence 39

3.2.8 Configurable Application Management 40

Chapter 4 Application Framework Infrastructure 41

4.1 Application Framework Components 41

4.1.1 Model 42

4.1.2 Presentation 44

4.1.3 Controller 45

4.1.4 Adapter 47

4.1.5 Coordinator 49

4.2 Compound Applications 53

4.2.1 Registering an Application as a Presentation 57

4.2.2 Registering an Application as a Controller 58

Chapter 5 Application Mapping 60

5.1 Application Generic Description 61

5.2 Application Customized Description 63

5.3 Mapping Mechanism 65

5.3.1 Manual Mapping 65

5.3.2 Automatic Mapping 66

Chapter 6 Application Management 69

6.1 Application Instantiation 69

6.2 Application Suspension and Resumption 73

6.3 Application Mobility 76

6.3.1 Intra-Space Mobility 76

6.3.2 Inter-Space Mobility 78

6.4 Application Adaptation 82

6.4.1 Manual Adaptation 83

6.4.2 Automatic Adaptation 84

6.5 Application Fault-Tolerance 86

Chapter 7 Active Space Applications and Computational Reflection 88

7.1 Computational Reflection and Open Implementations 89

7.2 Open Implementation Application Framework 90

7.3 Preserving the Application Semantics 91

Chapter 8 Framework Evaluation 93

8.1 Application Development Process 93

8.2 Building an Active Space Application without an Infrastructure 96

8.2.1 External Infrastructure Support 97

8.2.2 No External Infrastructure Support 100

8.3 External Evaluators 102

8.3.1 Understanding the Application Partitioning 103

8.3.2 Scope of the Application Framework 103

8.3.3 Complexity of the Application Framework 103

8.3.4 The Framework Allows Exploiting the Resources in the Active Space and Hides the Associated Complexity. 104

8.4 Application Performance 105

Chapter 9 Example Applications 109

9.1 Music Player 109

9.1.1 Application Components 109

9.1.2 Application Functionality Details 111

9.2 Presentation Manager 113

9.2.1 Application Components 113

9.2.2 Application Functionality Details 115

9.3 Ticker Tape 116

9.3.1 Application Components 117

9.3.2 Application Functionality Details 118

9.4 Location 119

9.4.1 Application Components 120

9.4.2 Application Functionality Details 121

9.5 PDF Viewer 121

9.5.1 Application Components 122

9.5.2 Application Functionality Description 122

9.6 Calendar 123

9.6.1 Application Components 124

9.6.2 Application Functionality Description 125

9.7 Attendance Manager 126

9.7.1 Application Components 126

9.7.2 Application Functionality Description 127

9.8 x10 Appliance Manager 127

9.8.1 Application Components 127

9.8.2 Application Functionality Description 128

9.9 Compound Applications 128

9.9.1 Ticker Tape Displaying Users' Location 129

9.9.2 Presentation Manager Controlling x10 Lamps 130

Chapter 10 Related Work 131

10.1 Active Spaces 131

10.1.1 Microsoft Easy Living....131

10.1.2 i-Land and Roomware....132

10.1.3 Stanford's Interactive Workspaces....132

10.1.4 CMU's Aura....133

10.1.5 Washington's one.world....134

10.2 Device Adaptation....135

10.3 Application Frameworks and User Interfaces....136

10.3.1 MVC, PAC, and ALV....136

10.3.2 BEACH....140

10.3.3 iCrafter....140

10.3.4 QCompiler....141

10.3.5 Pebbles....143

10.3.6 Graspable and Tangible User Interfaces....143

10.4 Computer-Supported Collaborative Work....144

10.5 Application Mobility....146

Chapter 11 Conclusions....148

11.1 Contributions....148

11.2 Perspective....149

11.3 Known Issues....149

Chapter 12 Future Work....152

12.1 Domain Specific Framework Extensions....152

12.2 Fault Tolerance....153

12.3 Application Mobility....154

12.4 Using Application Bridges to Alter the Meta-Level of the Applications....154

12.5 Extending the Application Framework to Support Multimedia Applications....155

References....158

Vita....174

List of Figures

Figure 1. Experimental Active Space....1

Figure 2. Physical vs. Active Space....2

Figure 3. Gaia Architecture....7

Figure 4. Lua Script Example: Application Instantiation and Assembly....19

Figure 5. Gaia Kernel Bootstrap Sequence....20

Figure 6. Application Framework Infrastructure....42

Figure 7. Model's Interface....43

Figure 8. Presentation's Interface....44

Figure 9. Controller's Interface....46

Figure 10. Example of a Controller Mapping Events from

a Mouse Controller, and a Context Controller....48

Figure 11. Controller's Interface....48

Figure 12. Coordinator's Interface....51

Figure 13. Inter - Application Communication Bridge....57

Figure 14. Application Bridge Interface....57

Figure 15. Registering an Application as a Presentation....58

Figure 16. Registering an Application as a Controller....59

Figure 17. Music Player AGD....63

Figure 18. Music Jukebox ACD for a Meeting Room Scenario (left)

and Music Jukebox ACD for a Home Scenario (right)....64

Figure 19. Application Mapping GUI....67

Figure 20. ACD Generator....67

Figure 21. ACD Generator Policy: Maximize....................................68

Figure 22. Music Player ACD....................................71

Figure 23. Application Instantiation....................................73

Figure 24. Suspension / Resumption Policy....................................75

Figure 25. Intra-Active Space Functionality Schematic....................................77

Figure 26. The Mobility Service Detects a User Leaving the Space

and Automatically Suspends All His/Her Applications....................................79

Figure 27. Resuming an Application....................................80

Figure 28. Application Builder GUI....................................84

Figure 29. Music Application Adaptation Based on the Number of People in

the Space....................................86

Figure 30. Music Player Adaptation Script....................................86

Figure 31. Preserving Application Semantics While Altering the Interactive

Configuration....................................92

Figure 32. Average Time to Instantiate the Presentation Manager Application....................................107

Figure 33. Average Time Required to Move a Presentation (slide)....................................108

Figure 34. Music Player Functionality Description....................................112

Figure 35. Presentation Manager Functionality Description....................................116

Figure 36. Ticker Tape Item....................................118

Figure 37. Ticker Tape Functionality Details....................................120

Figure 38. DCL Active Space Hierarchy (left) and

Corresponding Location Application Instance (right)....................................121

Figure 39. PDF Viewer Application Functionality Description....................................123

Figure 40. Calendar Application Functionality Description..125

Figure 41. Attendance Application Functionality Description..127

Figure 42. x10 Application Functionality Description...128

Figure 43. Ticker Tape and Location Compound Application and

Application Bridge Script...129

Figure 44. Presentation and x10 Appliance Manager Compound

Application and Application Bridge Script...130

Figure 45. Multimedia Application (left) vs. Non-multimedia Application (right)...................156

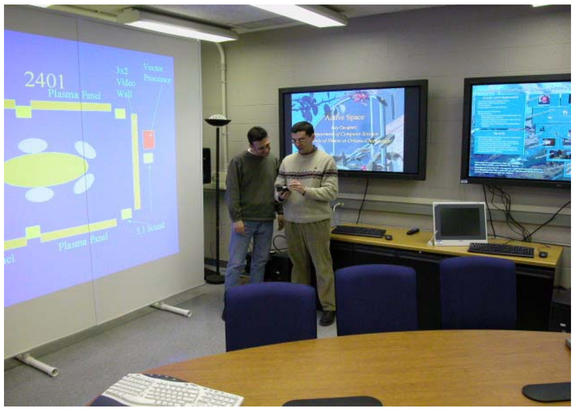
Introduction

"*The first step in innovation is to know that a thing can be created. After that, the rest is a matter of detail.*"

DUNE: House of Corrine.

Brian Herbert and Kevin J. Anderson

Pervasive computing environments encompass a spectrum of computation and communication devices that seamlessly augment human thought and activity with digital information, processing, and analysis to provide an observed world that is automated and enhanced by the behavioral context of its users. Large numbers of heterogeneous computing devices provide new functionality, enhance user productivity, and ease everyday tasks. In home, office, and public spaces, ubiquitous computing will unobtrusively augment work or recreational activities with information technology that optimizes the environment for people's needs.



15 Pentium IV at 1.2GHz
4 NEC HD Plasma Displays (61")
1 Sharp HD Projector
2 *Sound Web* Sound system
4 InfraredBeacons
2 iButton detectors
5 Touchscreen displays
4 Badge detectors
3 Compaq iPaq HandHeld devices
1 Gbit Ethernet network
1 Wavelan wireless network at 11Mbps
X10 appliance controllers

Figure 1. Experimental Active Space

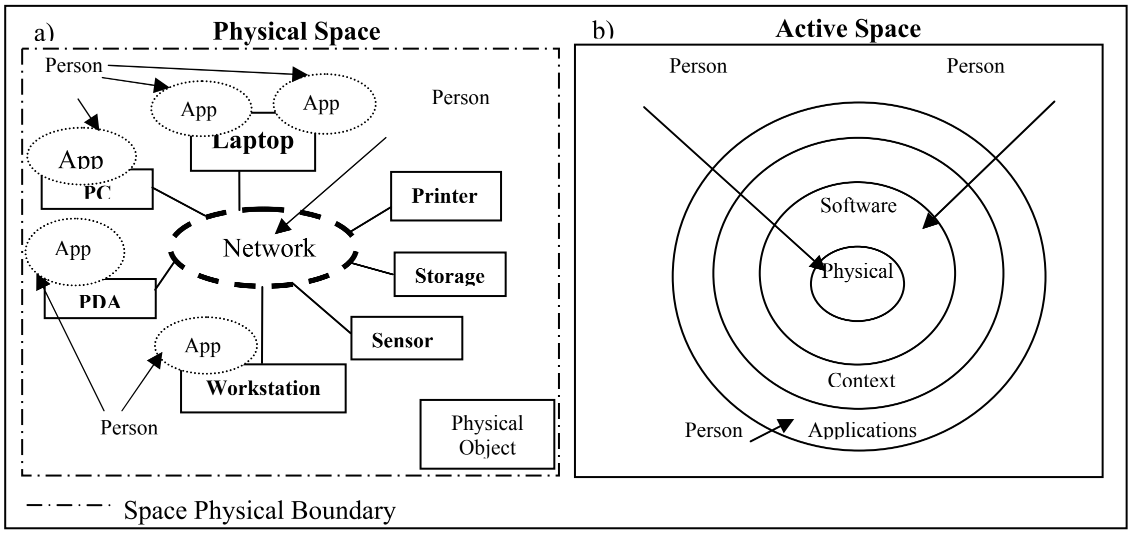


Figure 2. Physical vs. Active Space

The motivation behind this thesis is to develop a suitable software infrastructure to assist in the development of applications for ubiquitous computing habitats or living spaces. Figure 1 presents our prototype ubiquitous computing environment. In this environment, users interact with a number of devices simultaneously, register their own devices as a resource of the environment, require application adaptation according to changes in the environment, access data located in remote spaces, and suspend applications and restart them later either in the same place or in a different one. Homes, offices, and meeting rooms capable of sensing user actions and equipped with a large variety of devices will assist users with different tasks. We refer to these ubiquitous computing environments as Active Spaces, an extension to physical spaces. Physical spaces (Figure 2a) are geographic regions with limited and well defined physical boundaries, containing physical objects, heterogeneous networked devices, and users performing a range of activities. We define an Active Space (Figure 2b) as a physical space coordinated by a responsive context-based software infrastructure that enhances the ability of mobile users to interact and configure their physical and digital environment seamlessly. A requirement of

Active Spaces is to support the development and execution of user-centric, resource-aware, context-sensitive, adaptable, and mobile applications.

1.1. Long Term Vision

We believe that active spaces will become commonplace. People will interact with their physical habitats to customize their behavior, gain access to a greater amount of information, and benefit from the resources contained in the space. In this scenario, user data and applications are not confined to any one active space; they are associated with users using the notion of a “session”. This allows users to move across different active spaces and have their data and applications always available. When a user enters an active space, their sessions are dynamically mapped to the active space resources. Users can define different sessions and can activate and suspend them as required. We refer to the collection of sessions associated with a user and independent of particular active spaces as the *user virtual space.* This user virtual space requires support to locate resources available in the user’s environment and to map the sessions to the existing resources. The user-centric mobile application requirement of active spaces creates problems in our traditional approaches to building computer software infrastructure leading to a post-PC era of system software design.

Active spaces such as the one depicted in Figure 1 challenge existing assumptions for traditional PC applications. As observed by Marc Weiser [1], the problem raised by ubiquitous computing is to develop systems that vanish into the background. In an active space there is no longer the notion of a one-to-one relationship between a user and the interfaces of keyboard, mouse and display. Indeed, the complexity of ubiquitous applications encourages a relationship between a user and an active space. Active space system software support should simplify application programming and execution. In a similar manner to the role that operating systems play in supporting traditional PC applications, active space applications need support to access and operate the resources contained in the space that hosts their execution.

1.2. Thesis Scope

This thesis builds on the active space abstraction and describes a customized application framework, which includes basic building blocks and mechanisms to construct active space applications. The framework assumes the existence of a supporting software system infrastructure that effectively manages the active space.

According to [2], embedded computation is defined as follows: "*Computation will pass beyond desktop computers into every object for which uses can be found. The environment will be alive with little computations from computerized cooking appliances to lighting and plumbing fixtures to window blinds to automobile braking systems to greeting cards. To some extent, this development is already taking place. The difference in the future is the addition of networked communications that will allow many of these embedded computations to coordinate with each other and with the user. Human interfaces to these embedded devices will in many cases be very different from those appropriate to workstations.*" The application framework described in this thesis provides the building blocks and mechanisms to build applications for embedded computation environments.

This thesis contributes with the notion of active space, a novel active space meta-operating system, and an application framework to build active space-aware applications. These three contributions prove that it is possible to abstract the complexity of ubiquitous computing habitats and living spaces and present them as programmable environments. Furthermore, the thesis contributes with supporting services and tools to assist developers with the construction of applications.

Chapter 2 describes the systems software infrastructure we have built to manage active spaces (Gaia OS). This infrastructure is the foundation of the application framework. Chapter 3 presents an overview of the application framework. The application framework infrastructure is described in chapter 4, the specialization mechanisms in chapter 5, and the application management functionality in chapter 6. Chapter 7 introduces the concept of computational reflection and details how the application framework is implemented as an Open System. Chapter 8 includes a number of applications to validate the framework. Chapter 8 lists example applications built with the framework, including single and compound applications. Chapter 9 describes work related to this thesis, and chapter 10 concludes the thesis an presents future work directions.

Chapter 2

Gaia OS: An Active Space Meta-Operating System

Gaia OS manages the resources and services present in a physical space, and implements an application framework to assist users with the development of applications. It exports an application programming interface that hides the dynamism and complexity of the space, and allows developers to program the execution environment associated with the space. Gaia OS provides services for location, context, and events, and repositories with information about the active space. The system is built as a distributed object system. Figure 3 shows the three major building blocks of Gaia: the Gaia Kernel, the Gaia Application Framework, and the Applications

The Gaia Kernel contains a management and deployment system for distributed objects and an interrelated set of basic services that are used by all applications. The Component Management Core dynamically loads, unloads, transfers, creates, and destroys all the components and applications of Gaia. Gaia components are distributed objects and require communication middleware to support remote interaction. The current implementation of Gaia uses CORBA [3]; however, it is possible to port Gaia to other communication middleware architectures including SOAP[4], RMI[5], or customized implementations. CORBA provides a stable infrastructure for distributed object interaction. However, the dynamism and heterogeneity of active spaces require extensions to deal with issues such as soft-state to handle crashing components and resources added to and removed from the space. Gaia's five basic services are the Event Manager Service, Presence Service, Context Service, Space Repository Service, and Context File System. The Gaia software infrastructure implements a bootstrap mechanism that initiates the execution of the Gaia Kernel in any arbitrary physical space. Some of the services are built on top of existing middleware services (e.g. Event Manager), while others are extensions to the communication middleware (e.g. Presence Service).

Gaia applications use a set of component building blocks, organized as the Gaia Application Framework, to support applications that execute within an active space. The framework provides mobility, adaptation, and dynamic binding. The functionality permits commercial off the shelf as well as new applications to run in the active space. Active Space Application layer contains applications and provides functionality to register, manage, and control these applications through the Gaia Kernel services.

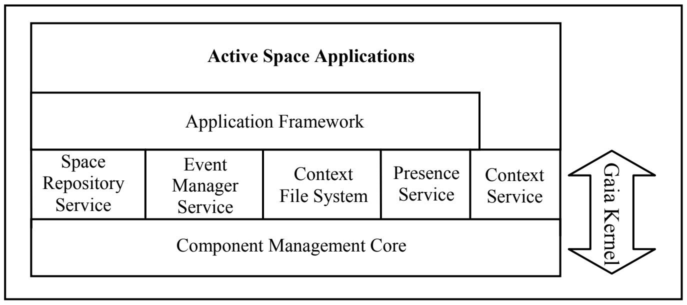


Figure 3. Gaia Architecture

2.1. Gaia Kernel

Gaia applications are component-based, distributed and mobile, and therefore require support for remote component execution and management. The Component Management Core (CMC) provides Gaia with functionality for component creation, destruction, and uploading. Remote execution nodes register with the active space and host the execution of Gaia components. The Gaia Kernel is composed of five services built as Gaia components which are described below in more detail.

2.1.1. Event Manager

Active Spaces are highly dynamic execution environments that require a flexible mechanism to expose changes in their current state. These changes include components starting, applications moving, users entering and leaving, component beaconing updates, and contextual changes. The Event Manager Service is responsible for event distribution in the active space and implements a decoupled communication model based on suppliers, consumers, and channels. Each channel has one or more suppliers that provide information to the channel and one or more consumers that receive the information. The event manager has a single entry point and one or more event channel factories. Each event factory is responsible for creating event channels with a specific behavior, e.g. high speed events, or persistent events. Gaia defines a default set of event channels to notify interested Gaia components about new services, applications, people, errors, and component heartbeats. Applications can also define their own event channels for application state changes. The event service allows applications to tap into event channels to learn about changes in the system.

The event manager is particularly convenient to decouple information suppliers from information consumers, therefore increasing system reliability. Suppliers and consumers can fail without disrupting the system. A crashing supplier can be automatically replaced with a replica that continues delivering messages to its assigned channel without the consumers being aware of the failure. Our current implementation of the event manager uses the CORBA Event Service [3] as the default event factory. More details of the event manager service can be found in [6].

2.1.2. Context Service

Gaia applications may use context information to adapt their behavior to accommodate user behaviors and activities[7]. Incorporating context into the infrastructure facilitates the adaptation of the computing environment to the needs of human users [8-11]. Our context service allows applications to query and register for particular context information so that they may adapt to their environment. The context infrastructure consists of a number of components, called context providers that provide information about the current context. These include sensors that track the location of people, the conditions within a room (e.g. temperature and sound) and other external conditions, such as weather and current stock prices. In addition, we also have components that can infer certain higher-level contexts based on sensed information. For example, we have a component that deduces the kind of activity going on in a room (i.e., meeting, presentation, or movie screening) based on who is in the room, which applications are running, and other cues. There exists a registry that maintains a list of the different context providers available and allows applications to find the providers that supply the contexts in which they are interested.

We use a model for context that is based on first order logic and boolean algebra, which allows us to easily write various rules to describe context information. These rules may be a combination of lower level context information. We represent context through a 4-ary predicate, whose structure is borrowed from a simple clause in the English language of the form <subject><verb><object>. An atomic context predicate is defined in the following way: Context(<ContextType>, <Subject>, <Relater>, <Object>). The Context Type refers to the type of context the predicate is describing, the Subject is the person, place or thing with which the context is concerned, and the Object is a value associated with the Subject. In our implementation, the ContextType is mapped to an event channel. The Relater relates the Subject and the Object such as a comparison operator (=, >, or <), a verb, or preposition. Some example context predicates are: Context(location, chris, entering, room 3231); Context(temperature, room

3231, is, 98 F); Context(social relationship, venus, sister, serena); Context(stock quote, msft, >, $60); Context(printer status, srgalw1 printer queue, is, empty); Context(time, , Is, 12:00 01/01/01). In some cases, one or more elements of a predicate may be empty (e.g., the Time context). It is possible to construct more complex contexts by performing first order logic operations such as quantification, implication, conjunction, disjunction, and negation of context predicates. One example of a rule is Context(Number of people, Room 2401, >, 4) AND Context(Application, Powerpoint, is, Running) => Context(Social Activity, Room 2401, Is, Presentation).

Ideas behind our context infrastructure have been inspired from the Context Toolkit [12]. We structure the expressive power of contexts with first order logic to frame rules and queries and to infer properties involving context using mechanisms that are similar to those of Prolog and other automated theorem provers. High-level context information may be determined from context information, similar to the aggregators of the Context Toolkit. Our system also formalizes the ways in which context information is exchanged between different components in the system and allows us to describe the properties of various components.

2.1.3. Presence Service

As a resource-aware infrastructure, Gaia needs to maintain updated information about resources present in the active space. The presence service is responsible for detecting digital (e.g., service and application) and physical entities (e.g., furniture and people) present in an Active Space. Our current implementation of Gaia defines four basic types of entities: Application, Service, Device, and Person.

The presence service implements a beaconing mechanism to maintain soft-state about entities present in the space and it is divided into two main subsystems: Digital Entity Presence and

Physical Entity Presence. The Digital Entity Presence subsystem detects digital entities; these entities periodically send heartbeats to notify the presence service that they are in the active space. When a digital entity fails to send the heartbeat, the digital entity presence subsystem assumes that the entity is no longer available – either it was stopped or it crashed – and therefore notifies the rest of the space that the entity left.

The Physical Entity Presence subsystem is responsible for detecting physical entities present in the active space. This subsystem uses different types of sensors to proactively detect the presence and, if possible, the location of physical entities. The physical entity presence subsystem implements the beaconing mechanism on behalf of the physical entities, acting as a proxy. The physical entity presence subsystem is implemented as an open infrastructure where different sensor device drivers and algorithms such as [12] can be incorporated. Our presence service differs from other existing context infrastructures (e.g. [12]) in that it provides functionality to detect and maintain soft-state information about software components, devices, as well as people.

2.1.4. Space Repository

Active space entities require functionality to learn about the properties of the resources available in the active space. For example, when an application starts executing, it uses the space repository to find appropriate resources (e.g., execution nodes, displays, and speakers). The space repository stores information about all software and hardware entities contained in the space (e.g., name, type, and owner) and provides functionality to browse and retrieve entities based on specific attributes. The space repository learns about entities entering and leaving the active space by subscribing to the channels defined by the presence service. Applications use the space repository during their instantiation to find suitable resources. This level of indirection allows us to describe applications in a generic manner (active space independent) and map them to the resources contained in different active spaces. All active space resources have an XML description associated that lists their properties (e.g., type and location). When new resources are introduced in the active space, the space repository contacts them to obtain the XML description and stores the information. The current version of the space repository uses a CORBA Trader [3] to store the data about entities. We are currently using the constraint query language defined by the trader, although we plan to extend this language in the future with a generic language that could be mapped either to the CORBA Trader constraint language or to standard SQL. For example, the following query: *Category = = 'Device' and Type = = 'Display'* returns a list of all displays present in the active space.

The Space Repository and the Presence Service implement the discovery protocol in Gaia. During the design of Gaia, our goal was to provide the minimum subset of functionality to enable resource discovery customized to active spaces. There are well known existing discovery protocols, but in most cases they do not take into account the geographic locality defined by an active space.

There are two aspects to discovery protocols: lookup and discovery [13]. Lookup is passive and assumes the existence of a directory service that clients use to find appropriate resources based on a query language. Discovery is spontaneous and assumes that the different entities in the system (i.e., clients, services, and directories) can found each other automatically. Services send information about their capabilities periodically, and clients and directories use these periodic broadcasts to find suitable services or store information about existing services.

Existing services such as LDAP [14] and CORBA Naming Service [3] are examples of lookup services. The main problem is that they do not implement any leasing or soft-state mechanism and therefore can end up with orphan references. This is not a desired behavior for a dynamic environment such as an active space, where resources can disappear without explicitly cleaning up.

Salutation[15], UPnP[16], and SLP[17] use multicast to advertise and discover services. The main problem is that in an active space there are resources that belong to different multicast domains (e.g., handhelds using Bluetooth and laptops using wireless connections). As a result, these resources would not be able to locate each other. Furthermore, multicast-based discovery protocols map the multicast domain to the discovery domain. That is, all resources belonging to a same multicast domain will be able to discover each other. This behavior is not desired in an active space, where the geographic boundary of the physical space (not the multicast topology) defines the computing environment domain.

Jini's[18] discovery protocol is based on multicast, although it allows contacting a specific Lookup Server (service that stores information about local resources). It allows creating different groups in the same multicast domain. One of Jini's main contributions is the use of leases to make the overall system robust without user intervention. However, the main drawback is the dependence on Java. It is not possible to use the discovery mechanism with other languages. Furthermore, Jini's querying capabilities (to locate specific resources) are not as rich as the ones provided by Salutation, SLP, or the Space Repository.

The Secure Service Discovery Service (SSDS) defined by the Ninja project[19] is also based on multicast, and therefore suffers from the same problems stated above (resources in different domains, and geographically bounded domains). Ninja, however, is a highly scalable and reliable protocol, and provides a high level of security not present in any of the other protocols.

In Gaia, discovery combines both lookup and discovery. The Space Repository is the lookup service and it uses soft-state to maintain the list of resources up-to-date. The Presence Service uses a soft-state mechanism based on heartbeats to detect resources entering and leaving the active space domain, and therefore implements the discovery mechanism. Resources in the Active Space use the Event Manager to send periodic heartbeats that are detected and processed by the Presence Service. The benefits of using events are two-fold. First, since every Active Space has its own collection of channels, events are not broadcasted to a multicast domain; instead, they are delivered to the domain defined by the Active Space. As a result, it is possible to have multiple Active Space instances in the same multicast domain. Second, event channels are independent of multicast domains; they are implemented as distributed objects that are accessible regardless of the network topology. As a result, resources assigned to different multicast domains can still coexist. The event channels define an overlay event distribution network, independent of the network topology.

2.1.5. Context File System

Active spaces are often designated for specific tasks. The context of these tasks can determine the information that is meaningful and can be used to prune out irrelevant material. Long running processes may not have the luxury of human intervention to locate required data, which may vary over time due to changes in context. If relevant information is known to exist in a particular location, the application is relieved from performing costly searches over the entire collection of data. In addition, users are highly mobile in active spaces and should not be burdened with manually transferring files or data, be it configurations, preferences, or application data from one environment to another. The environment should assist in making personal storage automatically available in the users' present location.

To address the foregoing issues, we have developed a context-aware file system (CFS) that uses context to alleviate many of the tasks that are traditionally performed manually or require additional programming effort. More specifically, context is used to 1) automatically make personal data available to applications, conditioned on user presence, 2) organize data to simplify the location of data important for applications and users, and 3) retrieve data in a format based on the context of user preferences or device characteristics through dynamic data types. CFS constructs a virtual directory hierarchy [20] based on the types of context that have been associated with particular files and aggregates related material. The layout of the directory hierarchy is implemented using a mounting mechanism, where mount points are owned by users and contain context tags. Users may merge their personal mount points into a space to make their data available to applications and other users. CFS is aware of different types of context, which are defined by the context service as well as by the users and applications.

Context is presented as directories, where path components represent context types and values. The file system path syntax uses the 4-ary predicate structure from the context service, where the relater defaults to equality. Context may be attached to files and directories by copying data to a context directory, which associates the particular context to the data. The virtual directory hierarchy forms a simple query language to determine what types of contexts are attached to files. For example, to determine which files have the context of location == RM2401 && situation == meeting associated with them, one may enter the /location:/RM2401/situation:/meeting directory.

CFS uses the current context properties of the environment (e.g., location, time, situation, weather) together with user specified properties to display the correct application data. For example, a seminar application may require all papers that are to be discussed during a seminar.

This application may be automatically started when the seminar is started, triggered from a calendar or when the moderator arrives, and therefore must be able to locate the correct files to display. The application simply opens the directory for the current papers, e.g., /type:/papers/current:. The file system will use the current location, situation, and time information along with the fact that "papers" are requested to find the correct files for the application. The contents of this directory may automatically change every week, as papers are added and old papers time out. However, from the application point of view, it simply opens the same directory every week and finds the relevant material.

The queries that are performed are not simply a combination of the current context and the application requested material; the space may define a context that is irrelevant to the current task. For example, the context "the weather is sunny" may be meaningless to the seminar application, but may make sense for a tele-presence application. The system resolves this issue by ignoring any context that is valid in the environment, but that is not explicitly associated with the data. Although the context directory structure is viewed as a hierarchy, context directories impose no fixed ordering, resulting in a forest rather than a tree structure; context paths can be traversed in any order.

The CFS architecture is composed of mount and file servers. Each active space has a namespace that is maintained by one mount server. The namespace changes as users physically move in and out of the space. File servers may be located locally or remotely to export storage to the local space. Our servers are implemented at application-level and leverage existing native file systems to access files and directories. More details about the CFS can be found in [21].

2.2. Application Framework

Active spaces entail a user-centric, resource-aware, multi-device, context-sensitive, mobile application model. Applications are partitioned among a group of coordinated devices [22], receive input events from different devices, present their state using different types of devices (e.g., sound system, display, and device to control temperature), and adapt to changes in the environment. Active spaces allow users to decide how to interact with applications using a number of inputs, outputs, and processing devices. However, development of applications for active spaces becomes a challenge. For example, an application may require moving the output data from one display to another, duplicating the output to different displays, transforming a visual representation into speech, and switching from a mouse to a voice controller, all of which must be performed providing application consistency guarantees.

Gaia provides an application framework with mechanisms to construct, run or adapt existing applications to active spaces. The framework is composed of a distributed component-based infrastructure, a mapping mechanism, and a group of policies to customize different aspects of the applications. The application framework infrastructure is based on the Model–View–Controller [23] pattern and introduces new functionality to export and manipulate the bindings of the application components; the mapping mechanism customizes applications to different active spaces; finally, the policies define different sets of rules to customize several aspects of applications including instantiation, mobility, reliability, and composition (number of components and their bindings). Chapter 5 presents a detailed description of the application framework.

2.3. Gaia Management Tools

Gaia relies on a scripting language to coordinate the digital entities running in an Active Space. This language simplifies the overall management of the active space. For example, we use a script to implement the bootstrap mechanism to start the execution of Gaia OS in a physical space**.** In this section we describe the scripting language we use in Gaia and the bootstrap mechanism.

2.3.1. Lua: Gaia's Scripting Language

Gaia uses a high level scripting language, called LuaOrb [24], to program and configure active spaces and to coordinate the active entities they contain. LuaOrb is based on the interpreted language Lua [25], which simplifies management and configuration tasks and allows for rapid prototyping and testing. The interpreter for Lua is fast and has a small memory footprint, which makes it suitable for resource-constrained devices. LuaOrb implements language bindings between Lua and CORBA, COM, and Java. The ability of LuaOrb to communicate with various component models directly allows it to easily interact with the components in our system. We use Lua to implement the bootstrap algorithm, to instantiate applications, to interact with execution nodes to create components and easily glue them together, and to quickly test components and applications.

Figure 4 presents an example script that instantiates and assembles an MP3 application. The script uses the Gaia space repository to obtain a handle to an audio output device (line 1), an execution node for the model (line 2), an execution node for the coordinator (line 3), and a touch screen called plasma 1 for the controller. Then, it uses the component management core functionality to create the coordinator (line 5), the model (line 6), the presentation (line 7), and the controller (line 8). Finally it assigns the model to the coordinator (line 9) and registers the presentation (line 10) and the controller (line 11) with the application using the interface exported by the coordinator.

```
1. local presentationExNode = Gaia.getEntity("Category == 'Device' and Type == 'AudioOut' ")
2. local modelExNode =          Gaia.getEntity("Category == 'Device'  and Type == 'ExecutionNode'
                                and Name == 'aspc1.uiuc.edu'")
3. local coordinatorExNode =   Gaia.getEntity("Category == 'Device'  and Type == 'ExecutionNode'
                                and Name == 'aspc2.uiuc.edu'")
4. local controllerExNode =  Gaia.getEntity("Category == 'Device'  and Type == 'Touchscreen'
                                and Name == 'plasma1'")
5. local coordinator =   coordinatorExNode:createComponent("Coordinator", "-name MP3Coordinator")
6. local model =         modelExNode:createComponent("MP3Model", "-name MP3Model")
7. local presentation = presentationExNode:createComponent("MP3Presentation", "-name MP3Player")
8. local controller = controllerExNode:createComponent("VCRInputSensor","-name MP3InputSensor")
9. coordinator:setModel(model)
10. coordinator:registerPresentation(presentation)
11. coordinator:registerController (controller)
```

Figure 4. Lua Script Example: Application Instantiation and Assembly.

Although the same result can be accomplished with languages such as C++ and Java, it requires more code, time, and user effort. Lua effectively simplifies the manipulation and coordination of entities.

2.3.2. Bootstrap

Gaia implements a bootstrap protocol that interprets a configuration file (Lua script) and starts the kernel services accordingly. The configuration file contains information about the Gaia Kernel services, such as the name of the service, the name of the interface of the service, the Gaia node or nodes that will host the service, the service instantiation policy (i.e., instantiate the service in all Gaia nodes or only in the first available Gaia node), and start parameters. Individual Gaia Kernel services can also specify additional configuration parameters. Currently, the active space administrator is responsible for providing the list of devices contained in the space. However, in the future we expect automatic device discovery by means of different sensor technologies.

Figure 5 illustrates a state transition diagram with the instantiation order of the Gaia Kernel services. Solid circles denote Gaia Kernel services, while dotted circles denote middleware specific services (CORBA in our current Gaia version).

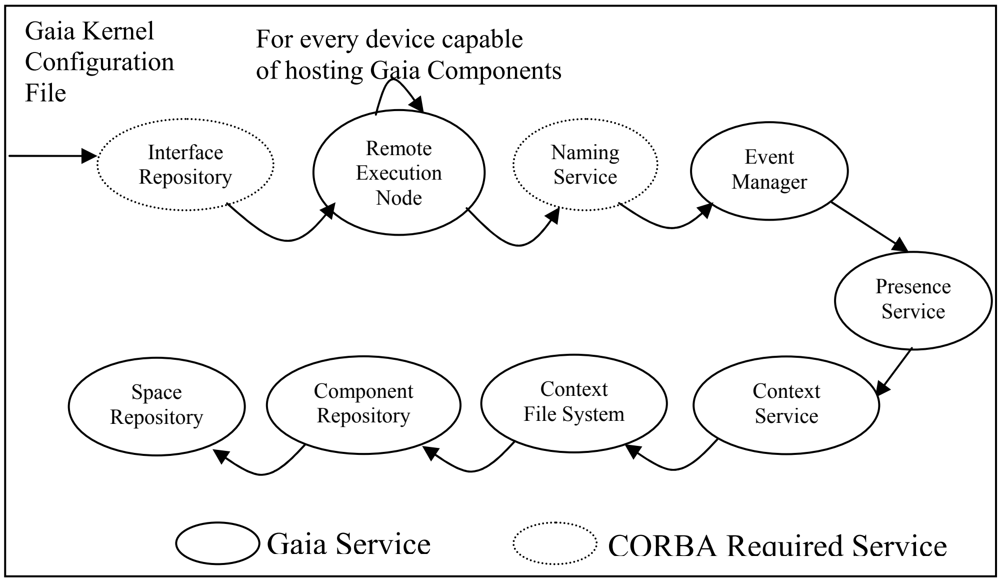


Figure 5. Gaia Kernel Bootstrap Sequence.

The configuration file contains a list of primary and backup Gaia Nodes for each Gaia Kernel service; the bootstrap process uses the dependencies diagram and the configuration file to decide whether or not to start a service in a particular Gaia Node. Gaia uses a timeout mechanism and a probing protocol to ensure that each Gaia Service is started in at most one machine (as specified in the configuration file).

2.4. Gaia OS and Traditional Operating Systems

The motivation behind Gaia OS is to abstract a space and all the resources it contains as a single programmable entity. However, this motivation is not new; it is the same motivation behind traditional operating systems. According to Silberschatz et al, "*the purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and* *efficient manner"* [26]. Gaia OS provides such environment at the space level (i.e. room, car, and home).

Silberschatz et al. define a group of seven services that are common for every operating system, namely: (1) program execution, (2) I/O operations, (3) File-system manipulation, (4) Communications, (5) Error detection, (6) Resource allocation, (7) Accounting and Protection.

Gaia OS provides functionality that covers the first six services defined by Silberschatz and implements a security service with functionality for accounting and protection. We provide next a comparison between the six traditional services and their Gaia OS counterpart.

Program Execution

The Gaia OS Component Management Core (CMC) provides functionality to create, destroy, and upload components in any execution node present in the active space. The CMC uses the program execution facilities of the execution node's OS, which includes memory, thread, and process management.

I/O Communications

Gaia OS leverages the low-level OS I/O functionality and provides device drivers (implemented as distributed objects) that export this functionality to the rest of the active space. Gaia OS also defines default I/O channels (i.e. input, output, and error), which are mapped to event channels. This allows creating a default "console" for the space.

File-System Manipulation

CFS provides functionality to manipulate files in active spaces. CFS interacts with the devices' low-level operating systems' file-systems to access and export the data to the active space. The specific location of the files is hidden from users and files can be accessed from any device in the active space. CFS extends traditional operating systems with functionality to transform data to different formats dynamically and to use context to organize the data based on different properties.

Communications

Gaia OS supports both direct and indirect communication mechanisms. Direct communication is similar to synchronous low-level OS IPC mechanisms, while indirect communication is the counterpart to asynchronous low-level OS IPC. Gaia OS provides RPC support for direct communication, and events (i.e. suppliers and consumers) for indirect communication. In both cases, Gaia OS leverages standard communication middleware. Events are similar to Unix signals and we use them in Gaia to notify entities in the active space about new resources added and removed, error conditions, changes in the file-system, and application state changes. The use of asynchronous events improves the reliability of the system by decoupling event producers from event listeners. While Gaia events are used mostly for signaling purposes, Gaia entities use other mechanisms such as RPC –for remote method invocations– and non-blocking streaming –for audio and video transmission.

Error Detection

Error detection includes both software and hardware errors that affect the execution of applications. Gaia OS uses the event service to report errors. Users register services that receive the error notifications and react accordingly (e.g. notify users, restart components, and suspend applications).

Resource Allocation

In traditional operating systems, resource allocation is related to the functionality required to manage hardware resources including memory, CPU, and disk. Gaia OS leverages this functionality and extends the notion of resource allocation to resources (i.e. devices, services, and applications) present in the active space.

The Gaia SR stores information about resources present in the space, their owner, status (i.e. free, used, available, and malfunctioning), and properties specific to each resource. Applications use the SR to obtain the resources they require to execute. Gaia OS implements the presence service that provides functionality that is conceptually similar to "plug and play" mechanisms offered by most modern operating systems. The heterogeneity and large number of resources contained in an active space require the presence services to maintain soft-state about existing resources for reliability reasons.

2.5. Gaia OS Evolution

Gaia OS is the result of six years of research on reflective middleware and meta-operating systems, middleware for handheld and embedded devices, and ubiquitous computing. Previous to the Gaia project, our group developed the 2K meta-operating system [27, 28], a reflective middleware operating system built on top of traditional operating system (e.g. Windows, Linux, Solaris, and PalmOS). 2K was strongly influenced by previous research on reflective middleware [29-31] and was built on top of a modified version of TAO [32], the pattern-based CORBA ORB from Douglas Schmidt et al. [33]. 2K hides device and operating system heterogeneity and can adapt dynamically to changes in the environment while maintaining the integrity of the overall system. Users in 2K interact with the system using different devices, therefore eliminating the one-to-one user-to-device traditional mapping. Individual devices in 2K become portals to the system. Following this approach, we started a new line of research to study how to integrate resource-limited mobile devices such as handheld and embedded devices into distributed computing environments [34]. These devices use middleware to interoperate with 2K and leverage the functionality provided by 2K services. As part of this research we developed an adaptable middleware prototype customized for handheld devices [35], which evolved into a fully reconfigurable middleware infrastructure[36]. This middleware allows bi-directional interaction between the handheld devices and the meta-operating system. As a result of our previous research, and influenced by the research of Georgia Tech [37-39], MIT [40, 41], and Berkeley [42] on ubiquitous computing, we created Gaia OS [43], a meta-operating system customized for physical spaces that supports the development of applications customized to these environments. Gaia OS provides a standard API that abstracts the complexity and heterogeneity associated with ubiquitous computing environments. The explicit binding between Gaia OS and physical spaces requires new services (not present in 2K) to take into account issues such as the context of the space and the detection of resources added to and removed from the space.

2.6. Gaia Evaluation

In this section we compare Gaia OS to other existing system software infrastructures for ubiquitous computing. This is a qualitative evaluation that compares the different infrastructures and how effectively they support the development of services and applications. The evaluation considers the aspects defined by Ponnekanti et al. [44]: Platform and Language Portability, Application Portability and Extensibility, and Recovery from Partial Failures. Furthermore, it adds two new aspects: Scalability and Explicit Application Development Support.

2.6.1. Platform and Language Portability

Ubiquitous computing environments are not isolated instances. The goal is to assume that every single physical space or habitat can potentially become an Active Space. Therefore, the supporting system software infrastructure must be portable across platforms and must be able to accommodate different programming languages. Gaia OS, iROS[44], and UPnP[16] are portable across platforms. However, Jini [18], One.World[45], are not due to their dependence on a single programming language (Java).

Gaia OS is based on a well-known communication middleware framework, CORBA, which runs on a large number of platforms. Furthermore, CORBA provides bindings to different programming languages, therefore guaranteeing language portability.

2.6.2. Application Portability and Extensibility

Application portability and extensibility refers to the complexity involved in integrating legacy applications into an Active Space, as well as the complexity to extend legacy applications so they can benefit from the multiple resources present in the Active Space. The iROS Event Heap allows integrating and extending existing legacy applications into active spaces without significant changes. Jini and UPnP do not provide support for application portability and extensibility, while One.World provides partial support. The Gaia OS Application Framework supports legacy applications. Based on the framework, it is possible to create Presentations that wrap existing applications as components of the application (i.e., viewers). The Presentations interact with the existing application via some well-defined mechanism such as Microsoft's COM or standard IPC, and the rest of the application components interact with the legacy application via the Presentation component. The application framework protocol allows multiple simultaneous Presentations. Therefore, if a legacy application has been wrapped as a Presentation, it is possible to synchronize multiple instances of the legacy application across several devices. As a result, application extensibility is a default behavior of the framework. For example, in Gaia OS, the Presentation Manager reuses Microsoft's Power Point to render the slides, and allows users to create an arbitrary number of viewers.

2.6.3. Recovery from Partial Failures

The iROS project is the closest to Gaia OS. They both consider physically bounded environments, and are built as middleware operating systems. The concept of geographically bounded spaces defines a limit on the scale of the project, which does not have to concern about environments such as the whole Internet, where the number of services and applications is several orders of magnitude larger. For this reason, I will discuss recovery issues related to Gaia OS and iROS, and will leave UPnP, Jini, and One.World out of the discussion, because they target different environments with different requirements.

iROS defines four recovery mechanisms: Level of Indirection (LoI) in communication, semi-persistent events, soft-state, and fast-recovery.

LoI is directly associated to the use of the Event Heap. Entities in iROS do not communicate directly; instead, they use tuples that are delivered by the Event Heap. If the target entity crashes, it is automatically restarted and continues receiving tuples from the Event Heap. The sender is not affected by the crash. Gaia OS uses a hybrid model that combines the use of events (provides decoupling) and direct peer-to-peer communication. The application framework provides mechanisms to reconnect faulty components of the application automatically, therefore automating the recovery of the overall application (transparent for the different components of the application). Regarding the kernel services, if any of them crashes, it is restarted and automatically registered in the directory service. Gaia OS provides a library that simplifies the access to the kernel services and hides remote communication mechanisms. This library can detect communication errors and automatically resolve the new kernel service instance.

iROS's Event Heap assigns a *time-to-live* value to the tuples so they are not immediately removed upon delivery. This behavior simplifies transient service failures. They simply restart and obtain the appropriate tuples. In Gaia OS, if a service crashes, clients interacting with it will receive an exception and will have to re-send the request, however, current Gaia OS implementation does not automate the process and therefore it is up to the clients. From the point of view of the application framework, if any of the interactive components (i.e. controller, or presentation) crashes, they are automatically restarted and reconnected to the model, and therefore they automatically synchronize with the model's state.

Both iROS and Gaia OS use soft state to improve the overall system reliability. Services and applications can leave the space, and as soon as they fail to send their beacon they are removed from the system.

Finally, iROS implements a fast-recovery mechanism that brings back the whole Event Heap when it crashes. This is the aspect where Gaia OS and iROS differ the most. In iROS, the Event Heap is the core of the whole system; event delivery and service communication require the Event Heap to function properly. If the Event Heap crashes, the whole system renders inoperable. Therefore, iROS requires a fast recovery mechanism that brings the Event Heap back quickly to minimize the disruption of the ubiquitous computing environment. The Gaia OS kernel is not built as a single entity that concentrates all the functionality. Kernel services are independent and distributed, and therefore can run separately. Furthermore, inter-service communication in Gaia is based on RPC, and therefore there is no intermediate entity that handles all communications. As a result, it is possible to have partial failures in Gaia OS but still keep the rest of the system running. For example, once an application has been created, if it does not require any kernel service, it is possible to stop the kernel and still use the application. The reason is that applications require the kernel to start, access files, and access the space repository. However, once an application is created, if it does not need any of those services, it becomes an autonomous unit.

2.6.4. Scalability

From the point of view of scalability, the distributed implementation of Jini, UPnP, One.World, and Gaia OS allows them to scale to large number of services. All these projects implement distributed services to handle different functional aspects of the system (e.g., service discovery and registration) and use peer-to-peer communication mechanisms that do not centralize all communications in a single service. On the other hand, iROS concentrates most of the functionality, including communication, into a single service, the Event Heap. The scalability of iROS is bounded by the processing and communication power of the machine that hosts the Event Heap and also by the number of services active in the ubiquitous computing environment. For example, the Event Heap is used to post service beacons, which are used for service advertisement. Also, all services in the system communicate via the Event Heap; therefore, communication intensive applications could overload the system or bring down its performance. As a result, as the number of services increases, so does the utilization of the event heap, which can become a bottleneck. While the Event Heap has proven effective for small environments such as offices and meeting room, scalability may become a concern for larger environments.

Gaia OS relies on peer-to-peer communication and on a distributed event system. This design allows distributing the load of the system and therefore simplifies scalability. For example, services send periodic heartbeat events to the presence event channel. The presence service listens to the channel, filters the events and sends two messages (service discovered, and serviced removed) to another channel called discovery channel. If the number of heartbeat messages goes beyond a certain threshold, the Gaia Event Manager can automatically create a new channel in a new host, use a load balancing algorithm to direct services to the appropriate event channel, and create a new presence service, therefore balancing the overall service announcement cost. Clients do not require any changes. They still use the same Event Manager to resolve the appropriate channel, and it is the Event Manager the one that implements the load balancing algorithm.

2.6.5. Explicit Application Development Support

To the best of my knowledge, Gaia OS is the first middleware implementation for ubiquitous computing environments (Active Spaces) to provide explicit support for application development (a framework for multimedia applications[46] and a framework for non-multimedia applications[47]). While all the other projects provide services that enable application construction, none of them provides tools or frameworks to simplify the development of applications that share common functional aspects (e.g., deployment, assembly, multi-device utilization, mobility, context-awareness, etc.). Based on our experience, providing explicit application development support reduces time and effort required to build applications significantly.

In order to explain the importance of the application framework, I will use the WWW model as an example. The WWW has evolved from a mere document distribution technology into a collection of distributed services that enable a broad range of applications (e.g. e-commerce). One of the mostly used techniques by most web sites is the dynamic generation of content and the utilization of backend services (e.g., web services, local programs, database interaction, etc). There are two widely accepted key technologies that enable this type of functionality: Java Server Pages (JSP) [48] and Java Servlets [49]. Java Server Pages allow creating HTML Web pages that contain code that is executed by the web server before the page is delivered to the client. Servlets are Java programs that are hosted by web servers, are invoked upon client requests, and generate the whole HTML web page dynamically. While JSPs allow reusing web page templates, as the amount of code embedded in the pages increases, it becomes too hard to maintain. The solution is to use JSPs and Servlets in combination. Servlets receive the original request, and based on certain properties they can decide what JSP to redirect the request to. Most web application developers use this technique to build dynamic and complex websites. However, manually developing applications in this way can become tedious. In order to solve this problem, there is a Web Application Framework (Struts [50]) that is based on the MVC pattern. This framework provides a set of abstractions and mechanisms that simplify the development of applications based on JSPs and Servlets. While it is possible to use JSPs and Servlets directly, it would take application developers more effort and time to get to the same results.

The goal of the Gaia OS Application Framework is the same as the one for Web Applications: provide specific abstractions and mechanisms that simplify the development of domain specific applications.

Chapter 3

Application Framework Overview

As described in chapter 2, an active space is the combination of a physical space, the resources it contains, its context properties, and a supporting software infrastructure into a single homogeneous programmable entity. An Active Space is the extension of the desktop metaphor to a physical space (user habitat). The space becomes the computer and users can benefit from it to perform a number of activities.

However, extending the concept of the desktop to an active space requires a novel application framework to develop, run, and adapt existing applications to the new environment. There are a number of issues that differentiate an active space from a traditional desktop computer:

1. **Multiple simultaneous input and output**. Traditional desktops are based on the window paradigm, where applications have a number of windows to display their output. These windows share a common monitor and therefore require a scheduling mechanism. This requirement applies also to input devices, which must be scheduled among all applications. Active spaces may have multiple input and output devices, and therefore, scheduling is not always required. It is possible to interact with different applications using multiple input and output devices simultaneously (e.g., speech recognition and a handheld for input, and three plasma displays and a laptop for output).
2. **User interaction with active spaces is not restricted to graphical interfaces**. Graphical user interfaces are, a subset of the possible interaction mechanisms. Some active space

applications do not have graphical representation. For example, an application that modifies the room temperature, does not have any graphical element. It receives the input from sensors present in the room or voice commands from the user and automatically adjusts the temperature level. The type of user interaction is not graphical, but instead perceptual. The application perceives changes in the environment (from sensors or user commands) and the user also perceives the output of the application (changes in the temperature of the room).

3. **The application model for an active space goes beyond the window/mouse paradigm.** Applicaton output is not restricted to a visual entity (normally a window). Instead, an active space application output can be any representation that affects human senses. Interaction with the application is not mediated by the graphical entity. Instead, controllers are attached directly to the application.

The application framework presented in this thesis accommodates the requirements of active spaces. It provides building blocks to construct perceptual user interfaces and exports functionality to customize applications to use multiple heterogeneous devices, alter the application composition dynamically, and construct collaborative and personal applications for arbitrary active spaces.

In the same way that an active space extends the notion of a desktop to a physical space, the application framework we propose extends traditional application building models to active spaces.

The problems addressed by the framework are: (1) defining an application infrastructure that can accommodate the requirements of active spaces including dynamically changing the cardinality, location, and quality of input, output, and processing devices used by an application;

(2) providing a mapping mechanism that allows defining applications requirements generically and automatically mapping them to the resources present in a particular active space; and (3) implementing a customizable application management interface to control different aspects of the application, including life-cycle and fault-tolerance.

3.1 Application Framework Scope

Applications built with the application framework share a common functional decomposition model based on input, output, and application logic, as defined by MVC. The framework is customized to simplify the construction of these applications (distributed component-based application model) and implements a number of protocols (e.g., instantiation, termination, and mobility) that automate the composition and life-cycle tasks required for Active Space applications. The application framework is generic enough to support the development of a large number of applications. However, the framework does not provide any functional support for specific types of applications such as multimedia and scientific applications. Instead, the framework must be specialized to support these specific application types.

This chapter describes the basic application framework with the default composition and life-cycle protocols. This basic functionality has proven sufficient to construct fourteen generic interactive applications that run in our prototype active space, and has demonstrated that it is possible to construct applications that combine multiple heterogeneous input and output devices. These applications are characterized by having a model (application logic) consisting of a single service, which contrasts with multimedia or scientific applications where the logic of the application is implemented by a collection of services that require specific synchronization and coordination algorithms.

While the framework can be used to construct other application types (e.g., multimedia applications), application developers would have to implement all required functionality. For example, let's assume a video on demand application that allows streaming video to different devices present in an Active Space. The application requires a number of different services (e.g., streaming service, encoders, decoders, and transcoders), as well as functionality to detect the resource utilization in the Active Space, functionality to compose the services dynamically based on resource utilization, and functionality to adapt the service composition as resource utilization (e.g., bandwidth) changes or new devices are deployed or removed from the Active Space. Based on the application framework, the model of the application would be implemented in a distributed fashion as a collection of the aforementioned services (i.e., streamer, encoder, decoder, and transcoder), and would also have to implement the algorithms to detect resource utilization and dynamically compose the services. However, this approach would make multimedia application development tedious. Application developers would be responsible for providing all functionality each time they build a multimedia application. Instead of forcing developers to provide all the required multimedia functionality, the application framework can be extended with the functionality described by Wichadakul et al. [46, 51] and Gu et al. [52, 53]. These projects describe a framework that assists in the development of multimedia applications for active spaces. The framework provides explicit support for multi-service model and implements algorithms to synchronize, coordinate, deploy, and monitor the services. As a result, developers can leverage the multimedia application development tools and services to quickly and effectively build multimedia applications for Active Spaces. I describe a possible extension of the application framework for multimedia applications in section 12.5. The proposed extension is based on the multimedia projects mentioned above.

3.2 Active Space Application Challenges

There are eight challenges I consider essential for active space applications. These eight challenges are the cornerstones of the application framework.

3.2.1 Resource-Awareness

Users in ubiquitous computing scenarios are surrounded by hundreds of devices including sensors, displays, and CPUs. In order for applications to exploit these resources, they must be aware of existing resources; the capabilities, availability, and cardinality of these resources. An active meeting room, for example, is aware of existing devices (e.g., plasma displays, projectors, servers, notebooks, PDAs, and sensors), services (e.g., light control, temperature control, and audio control), applications (e.g., collaborative document editor, and slide show manager), and people present in the meeting room, therefore allowing applications to exploit the resources. Efforts like UPnP[16] provide mechanisms to automatically detect and interact with devices and services present in the user local environment.

3.2.2 Multi-Device

In an environment where users are surrounded by hundreds of devices, the notion of logging into a single device becomes inappropriate. Users log into the active space (either implicitly or explicitly) and can take benefit of any entity contained in the space, as long as certain security and availability policies apply. This "post-pc" scenario requires a new model for application construction that allows partitioning applications into different devices as required by users and their associated context (e.g., time of the day, location, current task, and number of people). Application partitioning allows distributing functional aspects of an application (e.g., application logic, output, and input) across different devices. Remote terminal systems (such as X-Windows

[54]) allow redirecting the application output and input to different devices. However, they do not provide support to redirect the application output to one device and the input to another device. And for the same application, it is not possible to redirect multiple outputs to different devices. The type of application partitioning we seek is conceptually similar to the one described by Myers et al. [22], and provides fine grained control to choose a target device for each individual application functional aspect, as well as support for altering the application partitioning at run-time.

The application partitioning must be: (1) dynamic, so it may vary at run-time according to changes in the active space (e.g., new devices introduced in the space, or new people entering the space), and (2) reliable, in such a way that guarantees application integrity even when the application is distributed across different devices.

3.2.3 User-Centrism

Resource-awareness and the multi-device approach convey a third essential property: user-centrism. To accommodate application partitioning into multiple devices that vary over time, we bind applications to users and map the applications to the resources present in the space.

Abowd et. al.[55] use the term "everyday computing" to denote the type of applications associated with users that do not have a clearly defined beginning and end. Users may start these applications and use them for several days, months, or even years. Applications may be periodically suspended and resumed but not terminated. These applications are bound to users, and take benefit of the resources present in the users' environment.

User-Centrism requires applications to (1) move with the users, (2) adapt according to changes in the available resources (it may imply data format transformation, or internal application composition, or both), (3) provide mechanisms to allow users to configure the application according to their personal preferences, and (4) allow more than one user to participate in the same application.

3.2.4 Run-Time Adaptation

Active spaces are highly dynamic environments, where changes are the norm. Devices may be added to and removed from the space at any time, existing software entities may crash or new ones may be added dynamically, and users may enter and leave the space to start and stop participating in existing tasks. All these properties require applications capable of reacting to such changes at run-time. We consider two types of adaptation, functional and structural. Application functional adaptation (i.e. changing the behavior of the application algorithm) is an important feature that has already been applied to traditional applications by means of reflection [56] [57] [58] The requirement for dynamic application composition adaptation (altering the number and location of application components) does not apply to traditional interactive applications running on desktops due to, at least, three main reasons. First, the usage pattern for interactive desktop applications is different from the one observed in active space applications. Desktop users sit in front of the computer and use the local peripherals to interact with the application. If users move to a different computer, they restart the application or start a remote session (e.g. X-Windows, and Windows Terminal Services); it is not possible to split the application among several devices dynamically. On the other hand, active space applications' users are not bound to a single device; they can move freely around the space and use any available device; therefore, they expect the application to move and duplicate functionality to different devices dynamically.

Second, from an abstraction or granularity point of view, the desktop computer defines the execution environment, and therefore, there is no concept or need for splitting the application across different machines. However, in an active space, the active space itself (not the individual devices it contains) defines the execution environment (different abstraction granularities). Therefore, devices contained in the active space become execution nodes of a larger computing abstraction. From this perspective, applications require functionality to alter their composition dynamically to adapt to changes in the active space, and alter the application composition to use the most appropriate execution nodes according to user preferences and context parameters.

Finally, most interactive desktop applications are disconnected from external context attributes, and therefore, there is no need to adapt the application composition. The strong connection with context attributes in active spaces requires the application to adapt to new scenarios dynamically.

As an example of structural adaptation, consider a user reading a confidential document in an active office display. When the context of the active space indicates that a user is entering, the application moves the document to the user's personal PDA to protect confidentiality.

3.2.5 Mobility

Application partitioning and user-centrism require applications to be mobile. We consider two different types of mobility: intra-space mobility and inter-space mobility. Intra-space mobility is related to the migration of application components inside an active space and is the result of application partitioning among different devices. Intra-space mobility allows users and external services to move application components among different devices. The dynamic adaptive behavior of applications provides functionality for this type of mobility. Inter-space mobility concerns moving applications across different spaces, and is a consequence of the user-centrism (users are mobile by definition).

3.2.6 Context-Sensitivity

One of the main differences between an active space and a traditional distributed system is the utilization of the physical and digital context associated with the space as a default computational parameter. Context is one of the most important properties in ubiquitous computing [59] and therefore applications must have be able to access and alter existing context information. Context may trigger both functional and structural adaptation. As an example of functional adaptation, a news broadcasting application may select different types of news depending on who is in the room, the time of the day, or the mood of the users. And as an example of structural adaptation, a music application may use a user's laptop to play the music if there are other people present in the room; or may use the audio system of the room, the displays (to present the list of songs), and the room's speech recognition system to control the application when the user is alone.

3.2.7 Active Space Independence

Active spaces are characterized by containing a collection of heterogeneous devices. Furthermore, different active spaces have different number of resources. These two properties - heterogeneity and device cardinality – complicate the development of active space portable applications. Applications cannot make any assumption about the number and type of devices they will find in different active spaces. Traditional operating systems successfully address the issue of heterogeneity by providing software abstractions to represent the real hardware devices. However resource cardinality is not normally a concern in traditional operating systems, which can assume certain hardware configurations. For example, most personal computer operating systems can safely assume the existence of peripherals such as one monitor, one keyboard, one mouse, one audio device, one video card, and some storage device. Unfortunately, this does not apply to active spaces. While an active meeting room can have several devices such as displays, keyboards, and mice, an active car may not have any display, keyboard, or mouse. However, it may offer additional resources (e.g., speakers, and microphone) that make it possible to use the application prior to dynamic adaptation of the application.

Active space applications must be able to run in heterogeneous active spaces without requiring developers to customize the applications to each environment. Users should be able to use the same applications in their active home, active car, and active office.

3.2.8 Configurable Application Management

The dynamic nature of active spaces requires a configurable mechanism to control the management of applications. This management includes application lifecycle (instantiation, suspension, resumption, and termination), as well as other issues, including fault-tolerance and mobility. For example, there is no single application instantiation mechanism that fits all possible active space scenarios. Some scenarios may require that all application components must be properly created, in order to consider that the application is instantiated, while other scenarios consider that the application is active even if not all the components are successfully created. All application management aspects must be configurable so application developers can tailor them to the specific requirements of their applications. We believe that the application management mechanisms must be implemented in such a way that can accept user defined policies.

Chapter 4

Application Framework Infrastructure

The application framework we propose models applications as a collection of distributed components, and reuses the application partitioning defined by Model-View-Controller[23]. The framework exploits resources present in the application environment, provides functionality to alter the application composition dynamically (i.e., number, type, and location of the application components, as well as data format they manipulate), is context-sensitive, implements a specialization mechanism that supports the creation of active space-independent applications, and provides policies to customize different aspects of the application including mobility.

4.1 Application Framework Components

The application framework defines five elements: model, presentation (generalization of View), controller, adapter, and coordinator. The model, presentation, controller, and adapter are the application base-level building blocks and are strictly related to the application domain functionality. The coordinator manages the composition of the four base-level components and implements the application meta-level. It stores information about the composition of the application components and exports functionality to access and alter the component composition (e.g., attaching and detaching presentations and controllers, and listing current presentations). Figure 6 illustrates the framework UML and functional diagrams.

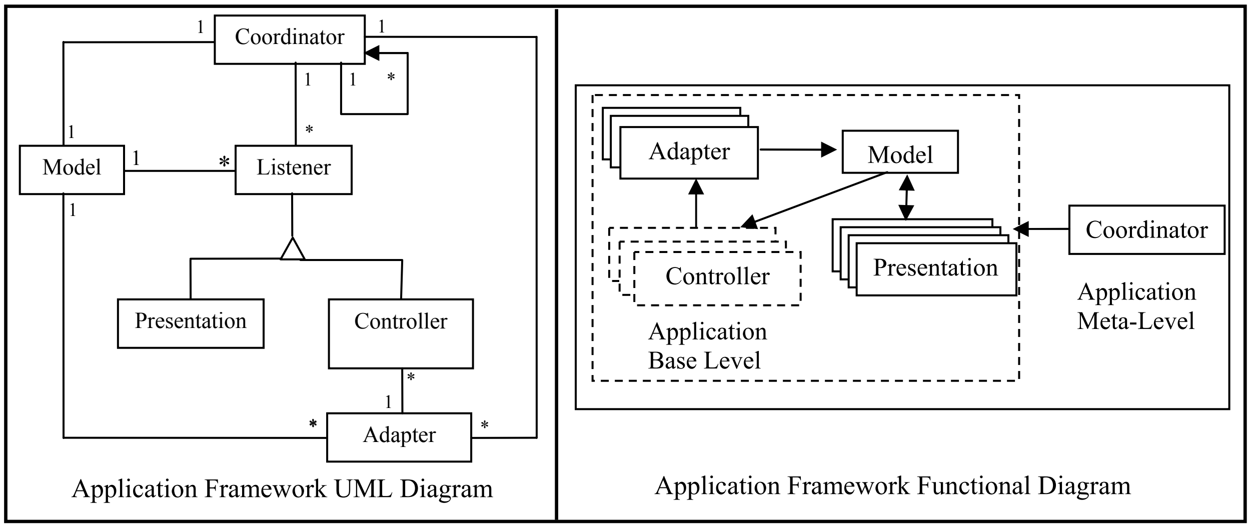


Figure 6. Application Framework Infrastructure.

4.1.1 Model

This component implements the logic of the application and exports an interface to access and manage the application's state. A model can be as simple as an integer with associated methods to increase, decrease and retrieve its value and representing a counter, or as complicated as a specific data structure with some related methods representing information about a document concurrently manipulated by a group of users. The model maintains a list of registered presentations and it is responsible for notifying them about changes in the application's state, therefore keeping all presentations updated. The application framework does not impose any restriction on the implementation of the model, which can be built as a single component or as a collection of distributed components.

The model's default implementation uses events to notify all attached listeners about changes in the application. When listeners receive a notification event, they examine the hint contained in the notification (e.g. a text message describing the nature of the change), and if required, contact the model to retrieve the new data. We use events as a control signaling mechanism to notify listeners about changes. Listeners use the most appropriate mechanism to retrieve the data from the model, including RPC, streaming, or any other mechanism required by the specific application. The application framework does not impose any specific communication mechanism between the model and the listeners.

Interface

Figure 7 lists the interface of the model, which consists of seven methods.

```
interface Model
{
   void attachListener(in string id, in Listener listener)
   void detachListener(in string id)
   void change(in any hint)
   void saveState ( in string path )
   void restoreState ( in string path )
   Coordinator getCoordinator();
   void setCoordinator(in Coordinator coord);
}
```

Figure 7. Model's Interface

The *attach* and *detachListener* methods are invoked to attach and detach components that get notifications when the state of the model changes. The default implementation of the model uses an event channel to notify listeners. However, it is possible to implement customized models that use alternative notification mechanisms. The *change* method is invoked by the model implementation to notify all attached listeners about a change in the application's state. The method takes an "any" parameter (a CORBA type that accepts any CORBA type) called hint that is received by all listeners. The value of this parameter is application specific and is helpful to allow listeners decide whether or not they need to contact the model to retrieve the new application state.

The application framework allows suspending and resuming applications, and defines two methods that models must implement in order to support such functionality. The model is the only component in the application framework that maintains state and therefore it is responsible for implementing functionality to persistently save it and restore it. When an application is suspended, the framework invokes the *suspend* method on the model, passing a Gaia Context File System's path [21], where the model can store its internal state. In the same way, when an application is resumed, the framework invokes the *resume* method on the model, passing the path pointing to the previously saved data. The last two methods (*set* and *getCoordinator*) allow accessing the coordinator of the application, which implements the meta-level of the application.

4.1.2 Presentation

The presentation transforms the application's state into a perceivable representation, such as a graphical or audible representation, a temperature or lighting variation, or in general, any external representation that affects the user environment and can be perceived by any of the human senses. The presentation generalizes the scope of the view component of MVC, which was originally defined as a graphical representation rendered on a display.

Interface

The presentation's interface is listed in Figure 8, and consists of seven methods.

```
interface Presentation
{
   void attachModel ( in Model theModel)
   void detachModel ()
   void notify (in any hint)
   void setId   ( in string presentationId )
   string getId ()
   Coordinator getCoordinator()
   void setCoordinator(in Coordinator coord)
}
```

Figure 8. Presentation's Interface**.**

Presentations are dynamically attached and detached to and from the model. When a presentation is attached to a model, the application framework invokes the *attach* method on the presentation and assigns the model's reference to the presentation. Presentations use this method as a constructor to obtain and present the application data when they are first attached to the model. When a presentation is detached from a model, the framework invokes the *detach* method on the presentation so the presentation stops presenting the application's data and returns used resources. All presentations must implement the *notify* method, which is invoked by the model whenever there is a change in the application's state (the *change* method in the model is invoked). There are scenarios where different presentations require presenting different aspects of the application. This is the case, for example, of the presentation manager application, which allows creating synchronized slide shows. Different presentations display different slides according to a user defined script. In this situation, the model notifications must contain an identification field to specify what presentations are affected by the change. The application framework allows assigning a unique id to presentations and retrieving the id, using the *setId* and *getId* methods. Presentations use the id to filter incoming notifications and decide whether or not they need to contact the model. The last two methods (*set* and *getCoordinator*) allow accessing the coordinator of the application, which implements the meta-level of the application.

4.1.3 Controller

We use the term controller to refer to any entity (i.e., hardware and software) capable of altering the application's state. Examples of hardware controllers are mice, keyboards, active badges, GUIs containing widgets that can be associated with user defined events, and context controllers, which are entities that process different context properties and synthesize specific context events that change the application's state. Using context as a controller has all the benefits described by

Salber et. al. [60], and simplifies the development of applications that can easily react to changes in the context.

Controllers receive notifications from the model so they can be synchronized with the application state. Controllers that do not require being synchronized with the model (e.g. array of push buttons and mouse) simply disregard the notifications. An example of a synchronized controller is the component used in the music player application developed using the framework. The controller displays a list of songs available. When the user selects one of the songs, the controller sends a request to the model to play the song. The music player allows adding and removing song titles dynamically and therefore requires the controller to be notified so it can update the list of songs it displays.

Controllers keep a reference to the model (to obtain data from the application), and a reference to the adapter (to alter the state of the application).

Interface

The Controller's interface has six methods and is listed in Figure 9.

```
Interface Controller
{
   void attachAdapter (in Adapter theAdapter);
   void detachAdapter();
   void attachModel (in Model theModel );
   void detachModel ();
   Coordinator getCoordinator();
   void setCoordinator(in Coordinator coord);
};
```

Figure 9. Controller's Interface.

We abstract controllers as distributed objects that alter the state of the application and are synchronized with the application model. These objects can be attached to and detached from the application dynamically according to user preferences or the context of the active space. When a controller is attached to the application, the framework invokes the *attachAdapter* method and stores a reference to the application adapter (a surrogate for the model described in the next section). Next, the framework invokes the *attachModel* method to store a reference to the application. Similarly, when the controller is detached, the framework invokes the *detachAdapter* and *detachModel* methods and removes the references to the adapter and the model. These four methods allow the controller to initialize and terminate their internal state when they are attached to and detached from the application. The last two methods (*set* and *getCoordinator*) allow accessing the coordinator of the application, which implements the meta-level of the application.

4.1.4 Adapter

This component coordinates the interaction between controllers and the model. Adapters map controllers' events into method requests for the model dynamically. This dynamic mapping mechanism allows reusing controllers with different models.

The adapter acts as a mediator between the controller and the model. Controllers invoke methods on the adapter and the adapter forwards the method calls to the model automatically. The adapter uses the mappings to change the name of the method invoked on the model dynamically.

The adapter's dynamic behavior allows customizing the behavior of the controllers according to context and user preferences. For example, the same controller running on a PDA could trigger different changes on the model depending on the user of the PDA (different user preferences) or the context of the active space (e.g. control the volume of the audio if the current context is a phone conversation, vs. selecting a new song if the context indicates we are listening to music).

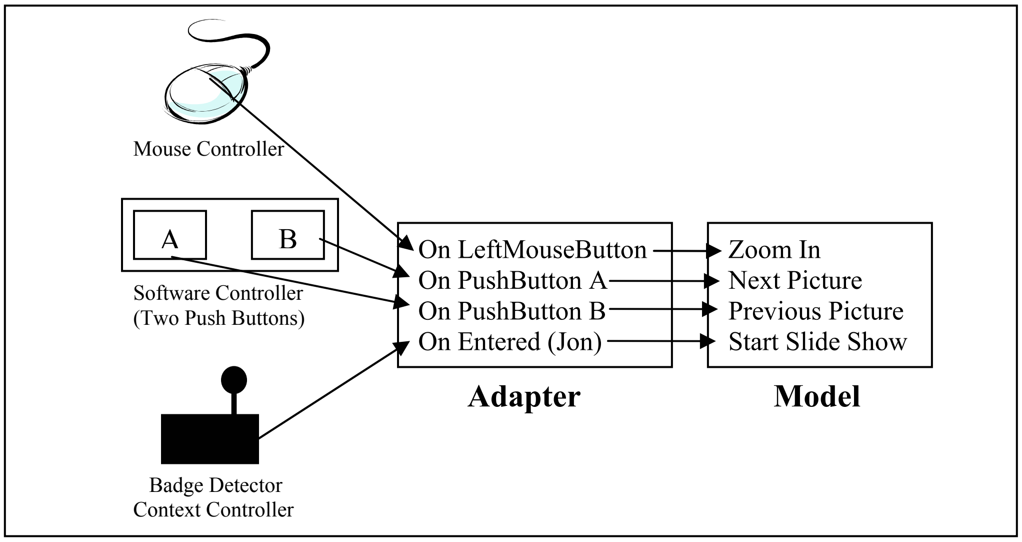


Figure 10. Example of an Adapter Mapping Events from a Mouse Controller, a Context Controller, and a Software Controller (GUI with Two Push Buttons) into Method Requests to the Model.

Figure 10 illustrates an example of an adapter translating the events received from three controllers into method requests for the model. The adapter's mappings can be set dynamically or can be specified in the script that launches the application.

Interface

The interface of the adapter consists of eight methods, listed in Figure 11.

```
interface Adapter
{
   void attachModel (in Model theModel);
   void detachModel ();
   void setMapping(in string original, in string target);
   void removeMapping(in string original);
   stringList listMappings();
   void controllerEvent(in string event, in any params);
   Coordinator getCoordinator();
   void setCoordinator(in Coordinator coord);
}
```

Figure 11. Adapter's Interface.

The application framework invokes the first two methods, *attachModel* and *detachModel*, to allow the adapter initialize and terminate its internal state when it is attached to and detached from the model. During the attachment, the framework assigns the model's reference to the adapter. The next three methods provide functionality to manipulate the mappings between controllers' events and models' methods. The *setMapping* method stores information about what method to call on the model (*target*) upon reception of a controller event (*original*). The *removeMappingMethod* provides functionality to remove existing mappings. Finally the *listMappings* method returns a list of all the mappings currently stored by the adapter. If the adapter receives a controller's event with no mapping defined, the adapter uses the name of the event as the name of the target method in the model. The *controllerEvent* method is invoked when controllers send an event to the adapter. However, controllers do not call this method directly; they simply send an event (method request) which includes the name of the method and a number of parameters to the adapter. The adapter parses the invocation, extracts the name of the event and internally calls the *controllerEvent* method, which uses the mappings' table to forward the request. The last two methods (*set* and *getCoordinator*) allow accessing the coordinator of the application, which implements the meta-level of the application.

4.1.5 Coordinator

Active space applications are a collection of distributed components composed of a model and a number of presentations and controllers. The dynamic nature of these applications challenges traditional interactive applications in terms of number and location of application components. In most of the cases, traditional interactive applications run in a single device and therefore those issues are not a concern. For an active space application, the number and location of presentations and controllers depends on the number of users, the nature of the space, and the activity taking place in the active space. After an active space application is started, it is common to add and remove presentations and controllers, or move these components to different devices contained in the space to better assist the user. As a result, active space applications require functionality to manage this particular behavior. The coordinator is responsible for implementing the functionality to support the particular behavior associated with active space applications. This functionality includes management of the application composition, fault tolerance, and lifecycle. Furthermore, the coordinator abstracts the distributed nature of the application to the users, and becomes the application management entry point.

The coordinator encapsulates information about the application components' composition (i.e., application meta-level) and provides an interface to register and unregister presentations and controllers. The coordinator provides also functionality to retrieve run-time information about the application's components composition, and allows for fine-grained control over the composition rules. This functionality does not exist in traditional MVC, where changing the application composition is not normally required. Applications for ubiquitous computing environments are subject to constant changes including resources present in the users' environment, availability of the resources, and changes in the application's context (e.g., location, time, number of people, and current task). The functionality provided by the coordinator allows applications to alter their internal composition to better adapt to changes. For example, a user entering an active office containing several plasma displays may want to move the calendar application presentation from his or her PDA to the active office. As a result, the application reconfigures itself to use all plasma displays to present different views of the calendar simultaneously (e.g., monthly, daily, and weekly view), and uses a touch screen, a keyboard, and speech recognition simultaneously to accept data and commands from the user.

The coordinator monitors the status of the application components and reacts to failures according to user defined policies. For example, if a component of the application stops running, the coordinator detects it and automatically unregisters the component from the application. This is the default policy, which can be overridden by users.

Interface

The coordinator's interface consists of twenty-six methods. These methods allow manipulating the meta-level of the application. Figure 12 lists the methods of the interface.

```
Interface Coordinator
{
   void setModel(in Model theModel)
   Model getModel()
   Adapter getAdapter(in string id)
   Adapter createAdapter (in string id)
   void deleteAdapter(in string id)
   stringList getAdapterIds()
   void registerPresentation(in Presentation thePresentation,
                 in string owner)
   void unregisterPresentation(in string id)
   stringList getPresentations(in string owner)
   void registerController (in Controller lis,
                in string owner, in string adapterID)
   void unregisterController(in string id)
   stringList getControllers(in string owner)
   void registerAppAsPresentation(in Coordinator presentation,
                in string script)
   void unregisterAppAsPresentation(in string id)
   stringList getAppPresentations(in string owner)
   void registerAppAsController (in Coordinator iSensor,
                in string listenerScript, in string isScript)
   void unregisterAppAsController(in string id)
   stringList getAppControllers (in string owner)
   void unregisterUserPresentations ( in string owner )
   void unregisterUserControllers ( in string owner )
   void terminateApplication()
   void setOwner(in string owner)
   string getOwner()
   void saveState(in string path)
   string generateCurrentACD()
}
```

Figure 12. Coordinator's Interface.

The *setModel* and *getModel* methods are used to set the application model and to obtain the application's current model. The coordinator creates an adapter by default with an empty mappings table. The *getAdapter* method returns a reference to the adapter identified by *id*, which can be used to manage the mappings. The *createAdapter* creates a new adapter that is assigned the id specified by the parameter. New adapters do not have any mapping. The *deleteAdapter* deletes the specified adapter. If there are controllers using the adapter, the method returns an exception. The *getAdapterIDs* returns a list with the ids of all adapters. The *registerPresentation* attaches a presentation to the application. The method stores information about the presentation and invokes the *attachListener* method in the model. Active space applications allow multiple users to be registered with an application. The *registerPresentation* method requires a parameter with the name of the presentation's owner. This information allows configuring security policies to determine the actions different users can perform on the application. Furthermore, the owner information is useful to disconnect all components of a user from an application with a single method invocation (e.g., user left the space). The *unregisterPresentation* method detaches an existing presentation from the application. The *listPresentations* method returns a list of all presentations attached to the application. Users can provide a user name to obtain only the presentations that belong to the user. If the parameter is a "*", all presentations are returned. The next three methods are similar to the previous three, but they are customized for controllers - *registerController, unregisterController, listControllers*. The last parameter of the *registerController* method (adapterID) is the id of the adapter that will adapt the controller's events. If no id is provided, the coordinator uses a default adapter.

The following six methods provide functionality to register applications as presentations and controllers, therefore allowing for recursive behavior. Section 4.2 provides detailed information about compound applications.

The next two methods – *unregisterUserPresentations, unregisterUserControllers* – allow unregistering all user components of a specific type (i.e. presentation and controller) from the application.

The *terminateApplication* method finishes the execution of the application by killing all application components. Even though applications can have components from different users, there is a user that owns the application. This user can define application policies and restrict access to other users. The application owner is specified at instantiation time as a parameter. The *setOwner* method can only be invoked by the current application owner to transfer ownership of the application to another user. The *getOwner* method returns the current application's user. The *saveState* method saves the current state of the application in the provided path. The coordinator forwards the request to the model of the application. The *generateCurrentACD* method creates a snapshot of the application and returns an Application Customized Description (ACD). This ACD stores information about the number of application components and their location. The ACD can be used to restart the application in the same status it was left. That is, the ACD will instantiate all components in the same location. This method is normally used after invoking *saveState*, and before calling *terminateApplication*. For more details about ACDs read chapter 5.

4.2 Compound Applications

The application components described in the previous section allow constructing active space-aware applications. However, each application implements a specific task (e.g. slide show management, playing music, and controlling appliances present in the space) and changes in one application do not affect other applications.

Active spaces export a physical space and the software and physical resources it contains as an integrated programmable environment. There are at least three different active space programming levels that must be considered: low-level, application-level, and active space-behavior level. The Gaia OS kernel provides functionality for the low-level programming, while the application framework infrastructure components provide the functionality for the application level. Programming the active space behavior requires an application composition mechanism to allow users to specify how changes on some applications affect other applications and therefore modify the behavior of the active space. For example, users should be able to specify that when the location application detects that the room is empty, the appliance control application should turn off the lights.

The application framework infrastructure provides functionality to create compound applications consisting of a number of interrelated applications – based on the diagram depicted in Figure 6, the Coordinator can be attached to many other Coordinators. This functionality follows a recursive scheme, where one application can be registered as a presentation or a controller of another. There are, at least, two different ways of implementing this functionality. The first approach uses inheritance to confer applications' models presentation and/or controller behavior, so they can be attached to other models. The second approach uses a component that allows defining external policies to specify application composition rules. Before explaining the pros and cons of each approach, keep in mind that an application developer should be unaware of composition issues. That is, a developer of a music application should not have to be concerned about the details of connecting the application to other applications. The main reason is that the developer does not know a priori what applications the music application will interact with. Application users are the ones who should have the ability to define arbitrary application composition rules.

The benefit of the first approach (inheritance) is that the model can be registered as a controller or presentation of another application directly, using the coordinator register presentation and register controller methods. However, this approach is too static. An application developer does not know a priori how the application will be used with other applications (presentation or controller). That is, some users may register the application as a controller of other applications, while other users may register it as a presentation (application developers cannot anticipate how their applications will be used in conjunction with other applications). However, using inheritance would require such knowledge, so the application can be registered properly. Furthermore, using inheritance would imply that the model should know how to react to notification messages from the application it is registered to. For example, if a ticker tape is registered as a presentation of a music application, when the music application sends a notification about a new song selected, the ticker tape should get the song name and display it. This behavior is not possible because it would imply that the ticker tape should have a priori knowledge of all possible applications it can be connected to. One possible solution would be to use external scripts that define how the application reacts to the notifications. These scripts are provided by the user composing different applications and can be attached to the applications dynamically. This solution still has a problem. Presentations and controllers can only be connected to one application at a time. However, an application running in an active space could be shared among several other applications. The ticker tape application, for example, could be used as a presentation of a music application, and a news application.

The second approach (external component) does not require the model to inherit from presentation and/or controller. The main benefit of this solution is that applications do not have to be aware of composition rules. This functionality is encapsulated in external policies and allows users to define arbitrary composition rules, therefore maximizing application reusability. This solution solves the two problems identified in the first approach: a priori knowledge and application sharing. Application developers do not need a priori knowledge of how the application will be composed with other applications. This knowledge is encapsulated in the external component. Regarding application sharing, adapters are associated with application pairs. Therefore, it is possible to connect an application instance to several other applications.

The application framework provides an application composition mechanism called *application bridge*[61], which implements the external component approach. This mechanism allows defining interaction rules among any pair of application instances running in the active space. The application bridge is generic enough to be used with any active space application regardless of the application functionality, it can be dynamically attached to applications without any a priori application knowledge, and it provides the means to easily customize the behavior of an active space by defining interrelation rules among applications. The application bridge is the mechanism to program the active space behavior, and it relies on external parameters such as context, application status, and user input to drive the application interaction rules.

The application bridge (Figure 13) is built as a controller that listens for notifications from the source application's model and introduces changes in the target application by invoking methods on the model via the adapter. The bridge implements functionality to execute user-defined rules that affect the state of the target application's model when it receives a notification from the source application. The functionality to invoke the script when receiving an event from the source applications is common to all bridges, and the rules defining what actions to take are bridge-dependent and are implemented as scripts (Lua) that are passed to the bridge at instantiation time. The script for the bridge receives a pointer to the source application's

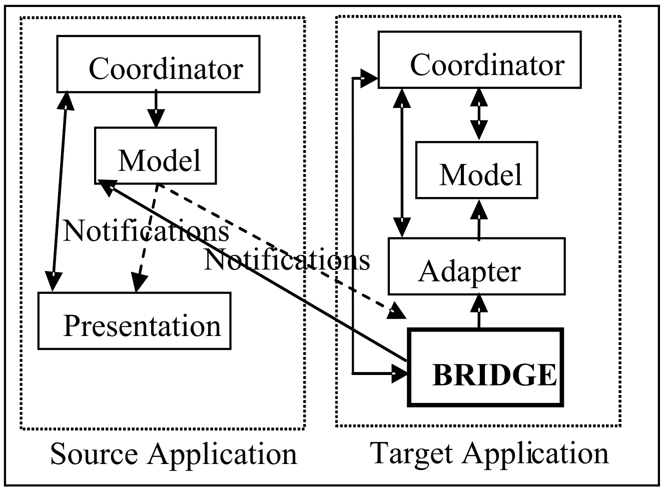


Figure 13. Inter - Application Communication Bridge.

model, a pointer to the target application's adapter, and the notification's hint (notification sent by the source application's model). Users write a script using these parameters to define the interaction rules. Figure 14 illustrates the interface of the script.

```
function(targetAdapter   , sourceModel, sourceEvent)
     <Composition rules>
end
```

Figure 14. Application Bridge Interface

The application bridge allows connecting an application as a presentation or controller of another application. This functionality abstracts applications as components, and recursively applies the composition rules defined by the application framework infrastructure (presentation, controller, and model). When an application A1 is registered with application A2, the application composition mechanism registers the coordinator of A1 with the coordinator of A2. This allows preserving application consistency. If A1 is terminated, A2's coordinator gets a notification and can remove A1 automatically. This is the default behavior implemented by the coordinator to manage the basic application components (presentation and controller).

4.2.1 Registering an Application as a Presentation

The coordinator's method *registerAppAsPresentation* provides the functionality to compose application A1 as a presentation of A2. This functionality includes creating a bridge from A2 to A1, registering A1's coordinator as a dependant of A2, and installing the application bridge script – provided as a parameter. Figure 15 illustrates the resulting composition. The application framework provides a default bridge script that simply forwards the source application's notification to the target application's model (invokes the change method in the target model).

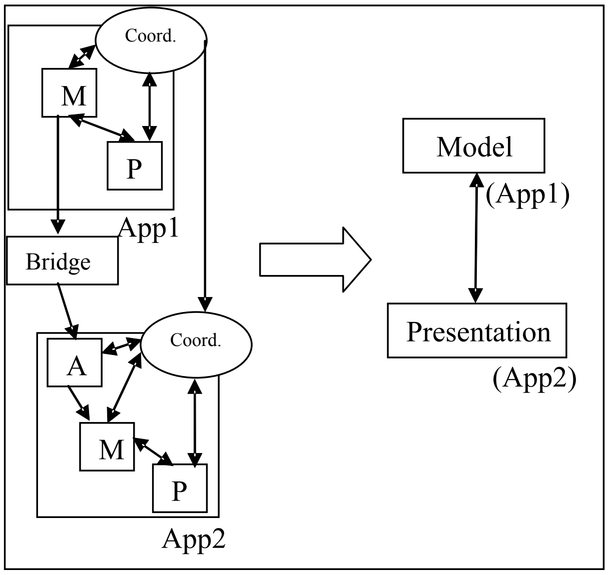


Figure 15. Registering an Application as a Presentation.

4.2.2 Registering an Application as a Controller

The coordinator's method *registerAppAsController* provides the functionality to compose application A1 as a controller of A2. This functionality includes creating a bridge from A2 to A1 (to receive the notifications), a bridge from A1 to A2 (to send the controller events generated by A1 to A2), registering A1's coordinator as a dependant of A2, and installing the two application bridge scripts – provided as parameters. Figure 16 illustrates the resulting composition.

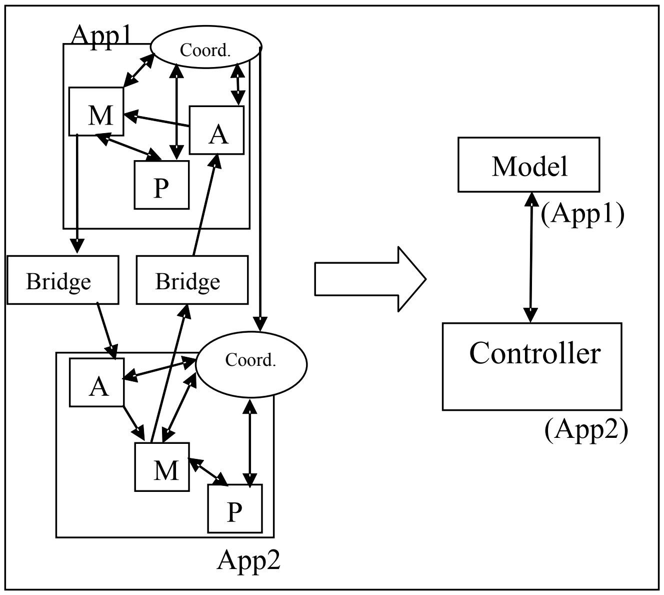


Figure 16. Registering an Application as a Controller

Chapter 5

Application Mapping

Traditional operating systems provide mechanisms to abstract hardware details from application developers so that applications built for a particular operating system can be used in machines with different hardware configurations. The application framework seeks the same type of functionality, that is, allow developers to build applications that can be used in heterogeneous active spaces. Users should be able to use the same applications in their active home, active car, or active office, without having to worry about different hardware configurations, and without having to modify the application to adapt it to new environments.

Active spaces are characterized by containing a collection of heterogeneous devices. Furthermore, different active spaces have different number of resources. These two properties - heterogeneity and number of devices – complicate the development of active space portable applications. Applications cannot make any assumption about the number and type of devices they will find in different active spaces.

Traditional operating systems successfully address the issue of heterogeneity by providing software abstractions to represent the real hardware devices and mapping those abstractions to the specific details of the existing hardware. However resource cardinality is not normally a concern in traditional operating systems, which can assume certain hardware configurations. For example, most personal computer operating systems can safely assume the existence of peripherals such as one monitor, one keyboard, one mouse, one audio device, one video card, and some storage device. Unfortunately, this does not apply to active spaces. While an active meeting room can have several devices such as displays, keyboards, and mice, an active car may not have any display, keyboard, or mouse. However, it may offer additional resources (e.g., speakers, and microphone) that make it possible to use the application prior to dynamic adaptation of the application.

Applications built with the application framework are independent of a particular active space by using generic application descriptions that list the application requirements. These descriptions are used to create a specific application description that includes resources present in the active space, which match the application requirements listed in the generic description. The framework defines two types of application descriptions: the application generic description (AGD), and the application customized description (ACD).

5.1 Application Generic Description

The AGD is an active space-independent application description that lists the components of an application and their requirements. The AGD contains a list of entries that describe each of the application required components. This list includes one model, one adapter, zero or more presentations, and zero or more controllers. Each entry contains name-value pairs to specify the component name, the number of component instances allowed, and the resources required by the component. These requirements include information such as for example, required operating system, and hardware platform. The mapping mechanism uses the requirements to query the active space infrastructure to obtain a list of matching resources. The syntax for the requirements is currently based on the CORBA's Trading Service query language [3]. The AGD is used as a template from which concrete application configurations (i.e., ACDs) are generated.

Figure 17 illustrates an example AGD for a Music Player application. This AGD includes one model, one coordinator, one presentation, and two controllers. The component that implements the model is called MP3Model, has a cardinality of exactly one, and a list of requirements that includes an execution node device running Windows2000. An execution node is any device present in an active space capable of hosting the execution of software components. The presentation component is called MusicPlayer (an audible presentation), has a cardinality of one or more component instances, and requires an execution node with audio output capabilities, running Windows2000. The first controller (SongSelector) displays a list of available songs and allows users to select one. It can have one or more active instances, and requires a touch screen execution node running Windows 2000 or Windows CE. The second controller (VCRInputSensor) provides functionality to start and stop the music, as well as functionality to move to the next and previous songs. It sends events to the model to alter its state. The component is optional because its cardinality is zero or more, and it requires a touch screen execution node running Windows 2000 or Windows CE. Finally, the coordinator requires exactly one instance and requires an execution node running Windows 2000.

```
Model
{
   ClassName  JukeboxModel
   Cardinality  1 1
   Requirements
       device=ExecutionNode
       and OS=Windows2000
}

Presentation
{
   ClassName   MusicPlayer
   Cardinality   1 *
   Requirements
    device=ExecutionNode
       and type=AudioOutput
       and OS=Windows2000
}

Controller
{
   ClassName   SongSelector
   Cardinality   1 *
   Requirements
   device=ExecutionNode
   and Type=TouchScreen
      and OS=Windows2000
      or OS=WindowsCE
}
```

```
Controller
{
   ClassName VCRController
   Cardinality  0 *
   Requirements
   device=ExecutionNode
      and type=Touchscreen
      and OS=Windows2000
      or OS=WindowsCE
}

Coordinator
{
   ClassName  Coordinator
   Cardinality  1 1
   Requirements
   device=ExecutionNode
      and OS=Windows2000
}
```

Figure 17. Music Player AGD.

5.2 Application Customized Description

The ACD is an application description that customizes an AGD to the resources of an active space. The ACD consists of information about what specific components to use, how many instances to create, where to instantiate the components, and who is the application owner. ACDs are implemented as Lua scripts [25] and they provide functionality to coordinate the instantiation and assembly of all the application components.

Figure 14 illustrates two ACDs generated from the AGD depicted in Figure 17. The ACD on the left is customized for a prototype active meeting room containing the equipment listed in Figure 1. The model is instantiated in host “amr1.as.edu”, and the coordinator in host “amr3.as.edu”. The ACD contains one Music Player presentation, which is run in “amr2.as.edu” because of its audio output capabilities. The SongSelector controller has two synchronized instances running in “plasma1.as.edu” and “plasma2.as.edu”. Finally, the VCRController is instantiated in a PDA.

The ACD on the right side of Figure 18 customizes the same Music Player application to an active home environment, which contains two desktops and PDA connected by a wireless and wired network, and managed by an active space software infrastructure (Gaia OS). The model, the coordinator, and the Music Player presentation are instantiated in “livingroom.as.home”. The song selector controller is instantiated in a PDA device, and no VCRController is created (it is an optional component).

```
Application =
{
 Model =
 {{
     ClassName =”JukeboxModel”,
     Hosts={{ “amr1.as.edu”}},
 }}
 Presentation =
 {{
     ClassName =”MusicPlayer”,
     Hosts={{“amr2.as.edu”}}
 }},
 Controller =
 {{
     Classname =”ListViewer”,
     Hosts={{“plasma1.as.edu”},
            {“plasma4.as.edu”},
           },
 }},
 Controller =
 {{
     Classname=”VCRController”,
     Hosts={{ “pda.as.edu”}},
 }},
 Coordinator =
 {{
     ClassName =”Coordinator”,
     Hosts={{“amr3.as.edu”}},
 }},
}

parseApplication(APPLICATION,
                 “mroman”)
```

```
Application =
{
 Model =
 {{
   ClassName="JukeboxModel",

Hosts={{"livingroom.as.home"}}
 }},

 Presentation =
 {{
     ClassName="MusicPlayer",
     Hosts={{"livingroom.as.home"}}
 }},

 Controller =
 {{
     ClassName ="ListViewer",
     Hosts={{"pda.as.home"}}
 }},


 Coordinator =
 {{
     ClassName="Coordinator",
     Hosts={{"livingroom.as.home"
}}
 }},
}

parseApplication(APPLICATION,
                 “mroman”)
```

Figure 18. Music Jukebox ACD for a Meeting Room Scenario (left) and Music Jukebox ACD for a Home Scenario (right).

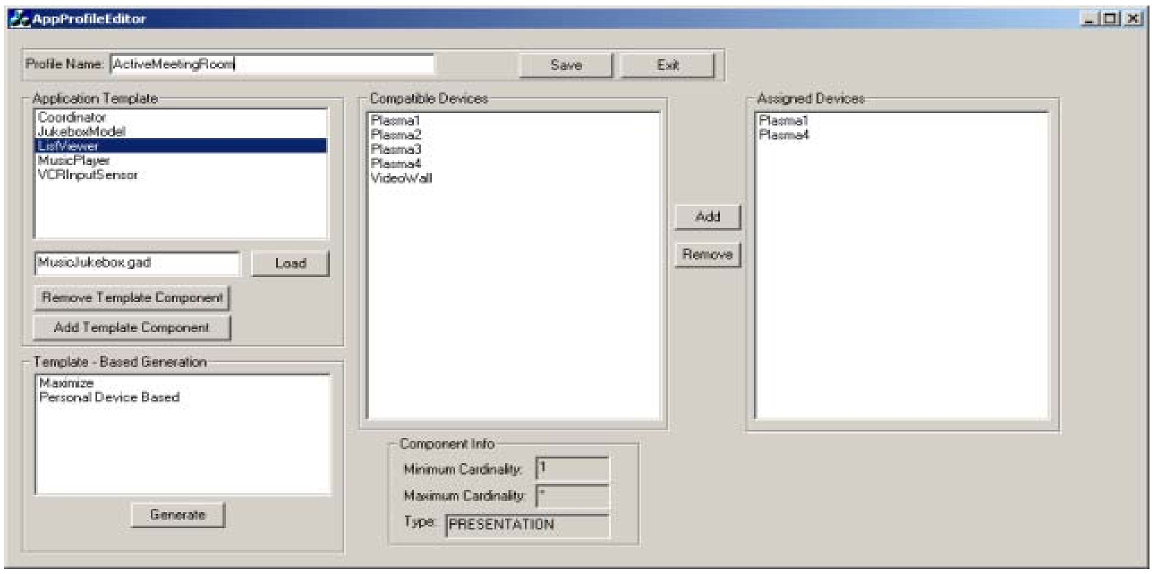
5.3 Mapping Mechanism

The mapping mechanism receives an AGD and a target active space, and generates and ACD customized for such space, according to a specialization policy. The diversity of resources present in an active space allow for multiple application configurations. This behavior contrasts with applications running in desktop computers where applications have a fixed number of resources. For example, the music player application presented in Figure 17 could be customized to the active meeting room with one to four song selectors running in any of the plasma displays, one to four VCR controllers running also in the plasma displays, and as many music player presentations as devices with audio output capabilities present in the space. If we also count the personal devices introduced by the users, the possible configurations are even larger.

The specialization mechanism offers two modes of operation: manual and automatic. In the manual mode of operation, users drive the specialization process by choosing the devices where the different application components are instantiated. The automatic mode of operation does not require user intervention and uses policies to drive the ACD generation process.

5.3.1 Manual Mapping

The application framework provides a graphical user interface (Figure 19) that parses an AGD and interacts with a user specified target active space to obtain a list of compatible resources for each application component. Users interact with the GUI to select the number of components and their location using the list of compatible resources. The GUI automatically generates an ACD based on the user preferences.



Profile Name: ActiveMeetingRoom

Application Template: Coordinator, JukeboxModel, ListViewer, MusicPlayer, VCRInputSensor

MusicJukebox.gad

Compatible Devices: Plasma1, Plasma2, Plasma3, Plasma4, VideoWall

Assigned Devices: Plasma1, Plasma4

Template - Based Generation: Maximize, Personal Device Based

Minimum Cardinality: 1

Maximum Cardinality: *

Type: PRESENTATION

Figure 19. Application Specialization Tool

The GUI allows registering templates that contain ACD generation rules. An example of such templates is the *maximized* template, which assigns one presentation to every compatible device present in the active space, limited by the maximum cardinality of the presentations. This semi-manual mechanism allows users to refine the resulting configuration before generating the ACD.

5.3.2 Automatic Mapping

The automatic mapping mechanism provides functionality to generate ACDs without user intervention. This mechanism uses policies to decide the number and location of each component and it is particularly useful to generate ACDs for new active spaces where user intervention is not possible or appropriate.

The application framework provides a service called *ACDGenerator*, which is responsible for the automatic ACD generation. The service allows registering Lua scripts that define the ACD generation rules. Figure 20 illustrates the ACD Generator, which provides an interface to register, unregister, list existing policies, and generate and ACD using a specified policy.

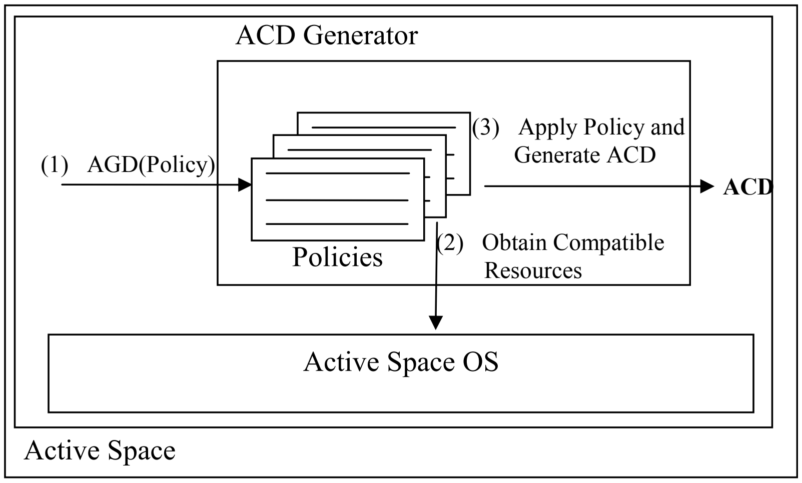


Figure 20. ACD Generator.

The ACD Generator parses the provided AGD, and passes the obtained information (i.e. number of components, names, cardinality, and requirements) to the specified policy. The policy contacts the active space OS and uses the information obtained from the AGD to obtain a list of compatible devices. Finally, the policy assigns devices to each of the components according to its own defined rules, and returns a list of assigned hosts. The ACD Generator combines the information obtained from the AGD with the list of hosts returned by the policy to generate the ACD.

Figure 21 illustrates an example policy called *maximize*. This policy assigns each presentation and controller to all available devices based on the presentation and controller's maximum cardinality. The model and coordinator are assigned to the first obtained compatible device.

```
if not Gaia then
  dofile( getenv("GAIA_ROOT").."/Boot/InitGaia.lua")
end

function assignHosts(table)
   i = 1
   while table[i] do
       --Obtain compatible devices
```

```
8.        local resources = Gaia.getEntityUCRs(table[i].requirements)
9.        local max;
10.       if table[i].maxCardinality == -1 then //-1 means infinite.
11.         max = resources.n //As many instances as devices available.
12.       else
13.          if table[i].maxCardinality >= resources.n then
14.             max = resources.n
15.          else
16.             max = table[i].maxCardinality
17.          end
18.       end
19.
20.       j = 1
21.
22.       table[i].assignedHosts.size = max
23.
24.       while j <= max do
25.          tinsert(table[i].assignedHosts, resources[j])
26.          j = j + 1
27.       end
28.       i=i+1
29.   end
30.end


31. do
32.    assignHosts(model)
33.    assignHosts(coordinator)
34.    assignHosts(presentation)
35.    assignHosts(controller)
36. end
```

Figure 21. ACD Generator Policy: *Maximize.*

The ACD Generator uses the AGD to create four tables called *model, coordinator, presentation,* and *controller*. Each one contains a list of components of the specific type with information about each component (name, cardinality, and requirements). The maximize policy defines a method called *assignHosts* (line 4) that receives a table and uses the requirements to obtain appropriate execution nodes (line 8), and the maximum cardinality to decide how many execution nodes to assign to each component (lines 10 to 18). Finally, the policy stores the name of the assigned hosts in one of the fields of the table (line 25). The ACD Generator uses the names contained in this field to generate the ACD.

Chapter 6

Application Management

This chapter describes the application management functionality, which includes management of the life cycle of active space applications (i.e. instantiation, suspension and resumption, and termination), mobility, adaptation, and fault-tolerance. Because of the dynamic nature of active spaces, there is no single algorithm for the different management tasks that fits all possible active space scenarios. I use policies (e.g., scripts, and services) that leverage the interfaces exported by the application framework to perform each of the management tasks. These policies can be as simple as a script that checks a simple condition (e.g., temperature) to decide, for example, how to adapt the application, or as complicated as an AI algorithm that adapts the application based on the user behavior. Policies allow users to customize each of the application management tasks according to their personal preferences, the nature of the active space, or the specific type of application. The use of policies allows also creating libraries with groups of policies customized to specific active spaces and tasks (e.g. active home, active office, and classroom assistant).

This chapter includes detailed information about the interaction between the application framework and Gaia OS in order to implement the management tasks.

6.1 Application Instantiation

Active space applications are a collection of distributed components, each one of them implementing some specific type of functionality (i.e. model, presentation, controller, and coordinator), and all of them targeting a common goal defined by the nature of the application.

Traditional applications are normally composed of an executable and an optional number of dynamically linked libraries, running in the same address space. When we run the application, the operating system loads the executable and all its required libraries into memory, and starts executing the program. The executable file contains a header with information about the required dynamically linked libraries. The operating system loader uses this information to load the libraries.

Active space applications are a collection of distributed components that interoperate using inter-process communication mechanisms such as RPC. A component is the smallest distributable execution unit in the system; it can have several formats, including an executable, a dynamic library, and a java class. Unlike traditional applications, active space application components do not necessarily share the same address space, or even the same machine. Therefore, they require a specialized instantiation mechanism capable of starting application components in any device present in the active space and responsible for assembling the components together.

The application framework ACD is the active space equivalent to an executable file. It contains information about the components required for the application, their names, initial parameters, and their target execution node. The application framework leverages the functionality provided by Gaia OS to instantiate the application components and to assemble them together. The instantiation algorithm is a policy, and can be customized for different applications or different scenarios. All instantiation policies must implement at least three steps: parse the AGD, instantiate the components in the execution nodes, interact with the coordinator to connect the components. The application framework provides two default instantiation policies implemented as Lua scripts: *strict* and *best-effort*. One of the benefits of implementing the policies as Lua scripts is that no additional functionality is required to parse the ACD (implemented as a Lua script table). The ACD can be imported and manipulated directly by the instantiation scripts.

Due to the distributed nature of active space applications, the instantiation mechanism must take into account the possibility of components crashing during the instantiation, and therefore must define what actions to take in case of failures. The *strict* policy guarantees that the application will be instantiated only if all components of the application are successfully created and connected. The *best-effort* policy guarantees that the application will be started if the model, coordinator, and at least one presentation and controller are successfully created and connected. This policy is useful in situations where the application has duplicated presentations, and therefore, if some of the presentations crash it does not affect the usability of the application.

Figure 23 illustrates the steps followed by the *strict* policy to instantiate the Music Player application described by the ACD depicted in Figure 22.

```
APPLICATION = {
   MODEL =
   {{
         CLASSNAME ="CORBA/MP3Model",
         HOSTS = {{ "desktop1",'-n MP3Model' },}
   }},
   PRESENTATION =
   {{
         CLASSNAME ="CORBA/MP3Player",
         HOSTS = {{ "desktop2","-n MP3Player" },}
   }},
   CONTROLLER =
   {{
         CLASSNAME ="Exec/VCRController",
         HOSTS = {{ "pda1","-n SongSelector" },}
   }},
   COORDINATOR =
   {{
         CLASSNAME ="CORBA/Coordinator",
         HOSTS = {{ "desktop1","-n MP3Player" },}
   }},
}
parseApplication(APPLICATION,"mroman")
```
Figure 22. Music Player ACD

The instantiation policy parses the ACD, obtains the description of the application components, and contacts the Gaia CMC (Component Management Core) to instantiate the component in the specified execution nodes (i.e. desktop 1, desktop 2, and pda 1). The CMC sends a request to the execution nodes to instantiate the assigned components. If no errors are found, the CMC returns the component references to the instantiation policy. In case of an error, the instantiation policy aborts the process (*strict* policy). This process is illustrated in Figure 23 a). After finishing the instantiation, the application itself does not exist yet. We simply have a collection of non-related components registered with the active space (Gaia Presence Service). The instantiation policy contacts the coordinator (using the previously obtained reference) and assigns the model (*setModel*), and registers the MP3Player presentation (*registerPresentation*), and the Song Selector controller (*registerController*). The coordinator automatically creates and registers a controller, and configures the application's internal state. This includes assigning the model's reference to the MP3Player, registering the MP3Player as a listener of the MP3 Model (dashed arrow from the model to the presentation), assigning the model's reference to the controller, and finally, assigning the adapter's reference to the controller. Similarly to the component instantiation step, if any error occurs during the assembly, the policy aborts the process.

The *best-effort* policy is really similar to the *strict* one. The only difference is that in case of errors, it checks if the model, coordinator, and at least one presentation and controller have been created. If they have, it continues with the process.

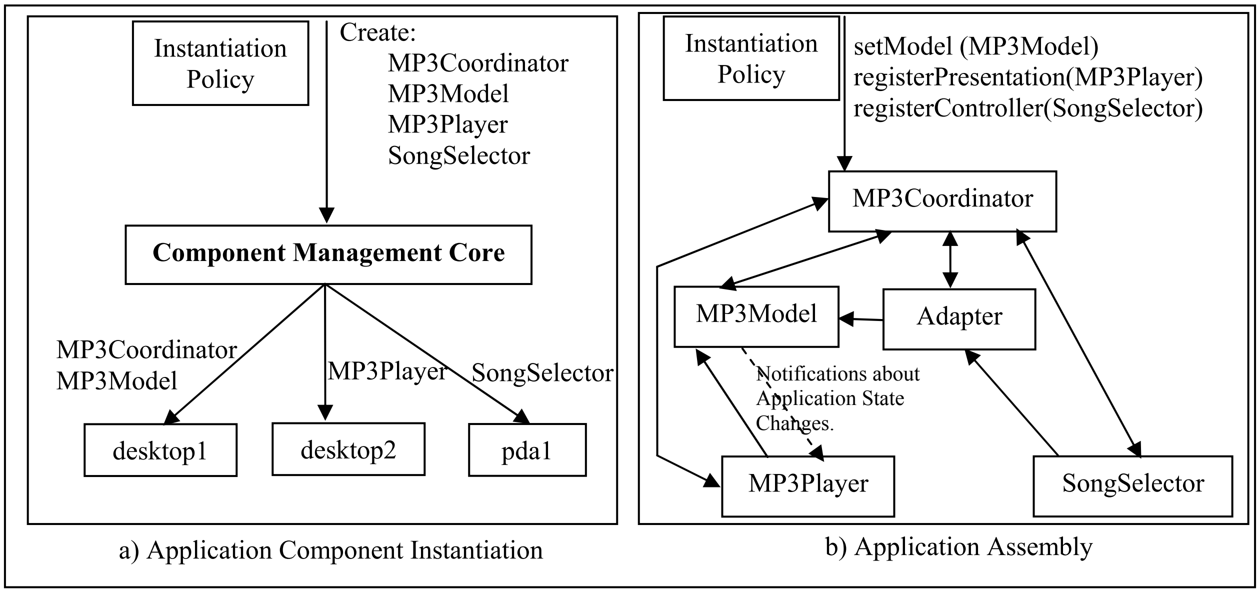


Figure 23. Application Instantiation.

6.2 Application Suspension and Resumption

The application framework's infrastructure defines that the model and the coordinator are the only two components that maintain state. The model stores state related to the functional aspect of the application (application base-level) while the coordinator stores information about the application composition (application meta-level). Presentations and controllers are both stateless. They obtain the state from the model.

The coordinator provides two methods to save the state of the application. The *saveState* method provides support to save the state of the application related to the application base-level. That is, the state relevant to the application functionality (e.g. current song being played, and volume). The default coordinator implementation forwards the request to the model of the application, which is responsible for saving the state in some appropriate format. The method receives a Gaia Context File System path[21], where it can save the data. This data can be accessed remotely from different active spaces. Saving the application state persistently is application dependent. The model is responsible for choosing a format to save the data, as well as for deciding what data must be stored. The application framework does not provide any additional support. Future implementations could include a specialized model that provides serialization support. The second method related to state saving is called *generateCurrentACD*, and it provides functionality to generate an ACD that matches the current application layout, including the number of components, their location, their names, and the persistent id's assigned to the presentations (if any). The returned ACD can be used to re-instantiate the application so it creates the same number of components, and in the same locations. The ACD is only useful if the application is resumed in the same space where it was suspended. Otherwise, the ACD can be used to learn about the number of components the application had before it was suspended, and negotiate with the new space to find appropriate new resources. This would be the task of a specific instantiation policy. Suspending a compound application requires forwarding the request to the applications attached to the current application, which will recursively follow the same steps.

The application framework provides a default policy to suspend and resume an application in the same space. In order to suspend and resume an application in different active spaces, an additional mobility infrastructure is required. For more details about such infrastructure read [62]. Figure 24 illustrates the steps followed to suspend and resume an application; the functionality is provided in an external library.

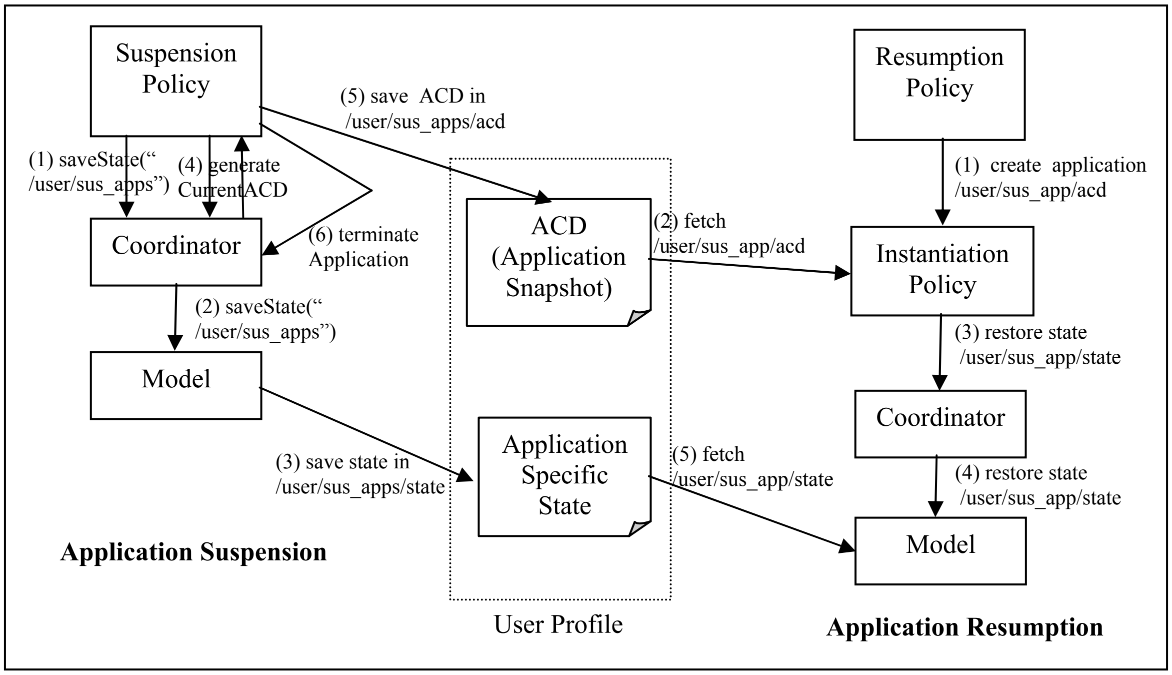

Suspension Policy
(5) save ACD in /user/sus_apps/acd
(1) saveState(" /user/sus_apps")
(4) generate CurrentACD
Coordinator
(6) terminate Application
(2) saveState(" /user/sus_apps")
Model
(3) save state in /user/sus_apps/state
Application Suspension
ACD (Application Snapshot)
Application Specific State
User Profile
(2) fetch /user/sus_app/acd
(5) fetch /user/sus_app/state
Resumption Policy
(1) create application /user/sus_app/acd
Instantiation Policy
(3) restore state /user/sus_app/state
Coordinator
(4) restore state /user/sus_app/state
Model
Application Resumption

Figure 24. Suspension / Resumption Policy

In order to suspend the application, the policy sends a request to the coordinator to save the application's state, and passes a path where the data can be saved (1). The coordinator forwards the save state request to the model (2), which uses the functionality from the Gaia CFS to store data relevant to the application base-level (3). Next, the policy sends a second request to the coordinator to generate an ACD for the current application composition (4). The policy uses the Gaia CFS to store the ACD persistently (5). Finally, the policy terminates the application (6). Note that for compound applications, the default policy does not forward the terminate application request to the registered applications. The reason is that these applications might be shared among a number of other applications. Different policies, though, can modify this behavior.

When the user sends a request to resume the application, the policy uses the instantiation functionality to restart the application, using the ACD that was previously stored (1,2). Next, the policy sends a request to the coordinator to restore the state of the application, passing the path where the state was saved (3). The coordinator forwards the request to the model (4), which uses the path to obtain the previously saved state (5). At this point, the application resumes the execution.

Note that by using an adaptive instantiation policy (a policy that examines an ACD and adapts it to the resources of a different space) it would be possible to suspend the application in one space, and restore it in another one.

6.3 Application Mobility

According to section 2.5, there are two different types of application mobility: intra-space mobility, and inter-space mobility. Intra-space mobility allows moving components of an application inside an active space. On the other hand, inter-space mobility is related to moving an application (the entire application or a subset of components) among different active spaces. This type of mobility requires application adaptation to deal with active spaces with different type and number of resources.

The Gaia Application Framework provides functionality to build applications that support both inter and intra-active space mobility [63].

6.3.1 Intra-Space Mobility

Intra-space mobility allows moving the interactive components of an application (i.e. presentations and controllers) to different devices present in an active space. We assume that the model, coordinator, and controller do not move in an active space – although it is possible to do so using the inter-space mobility functionality.

The application management interface provides support for mobility. The functionality is implemented as a library that interacts with Gaia OS to find compatible devices, and with the coordinator of the application to move the components. Figure 25 illustrates a schematic of the intra-space mobility functionality.

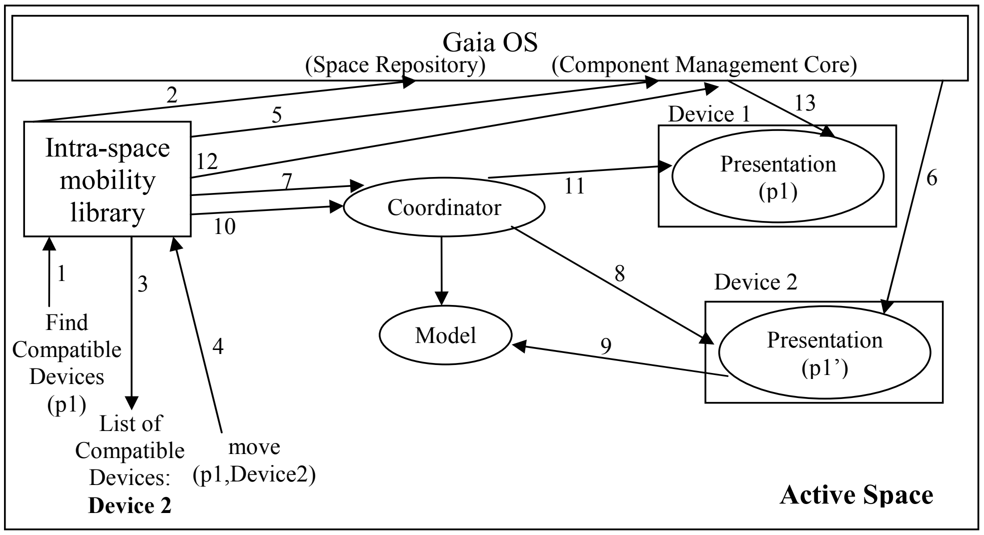


Figure 25. Intra-Active Space Functionality Schematic.

Moving a presentation or a controller (we use a presentation as an example) starts by obtaining compatible devices for the selected component (1). The library retrieves the component requirements (XML description) and contacts the Gaia OS Space Repository (SR) to obtain a list of compatible devices (2). The SR is a database of entities present in the active space. It stores entity references with both static and dynamic information about the entities. The static information includes information such as type of component and category, while the dynamic information is optional and specific to each component. For example, the dynamic information associated with an execution node includes CPU utilization and bandwidth availability. Gaia OS provides a set of QoS services [51] to monitor resource utilization. The dynamic information is obtained from these services.

The SR returns devices that match the component requirements (3). Users choose a device and invoke the move method in the library (4). This method contacts the Gaia OS Component Management Core (CMC) (5) to create the component in the device selected (6). Next, the library sends a request to the coordinator to register the new component p1' (7). The coordinator stores the presentation's reference, registers it with the model, and notifies the presentation it has been attached to an application (8). The presentation contacts the model (9) and obtains the application state, therefore synchronizing itself with the application. Once the new presentation is properly registered, the library contacts the coordinator and sends a request to unregister the original presentation (10). The coordinator notifies the presentation it has been detached from the application (11), therefore allowing the presentation to release resources. Finally, the library uses the Gaia CMC (12) to delete the original presentation (13).

The mechanisms described in figure 4 are common to every application based on the application framework. Therefore, application developers can leverage the mobility mechanisms and concentrate on the application functionality.

6.3.2 Inter-Space Mobility

In an active space, applications are not confined to a single machine. Applications exploit several devices simultaneously and therefore run in the context of the active space, which can be abstracted as an execution node. However, the application-to-active space association is temporal and changes when the user that owns the application moves to a different space.

Inter-space mobility provides functionality to move applications automatically as users roam across active spaces. This functionality is implemented as a service that monitors applications and users in active spaces. Every active space runs an instance of the service. The type of mobility we describe in this section assumes that all application components move to the new space[1].

One of the issues with inter-active space application mobility is that different active spaces have different resources. As a result, the structural composition of an application may not remain constant from one space to another. The mobility mechanism relies on the application specialization mechanism to adapt the application to the new scenario.

Inter-space mobility leverages the application management interface for application suspension and resumption. Moving an application requires first suspending the application, and then resuming the application in a (possibly) different active space.

Suspending the Application

Figure 26 illustrates the steps required to suspend an application when a user leaves an active space. When a user leaves, the inter-space mobility service receives a notification (1), retrieves a list of applications owned by the user and suspends them (2). While this is the default behavior, users can configure what applications should move.

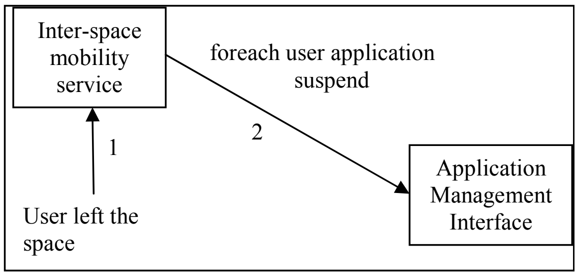

Inter-space mobility service
foreach user application suspend
2
1
Application Management Interface
User left the space

Figure 26. The Mobility Service Detects a User Leaving the Space and Automatically Suspends all His/Her Applications.

[1] For information about partial application mobility read [8].

Resuming the Application

The resumption policy described in section 6.2 assumes that the application is suspended and resumed in the same space. Therefore, the policy reuses the ACD that was generated during the application suspension. The mobility policy implements a new resumption policy that can resume an application in an active space different from the one where it was suspended.

Existing mobility solutions [64] [65] [66, 67] assume that the structure of the application does not change when the application moves. However, in an active space, the assumption is no longer valid. The target active space may differ from the original one in number of resources, type of resources, QoS properties, and security parameters, and therefore, the application structure (composition) may require adaptation. The mobility service implements a resumption policy that negotiates with the target active space to adapt the suspended application. The process is illustrated in Figure 27.

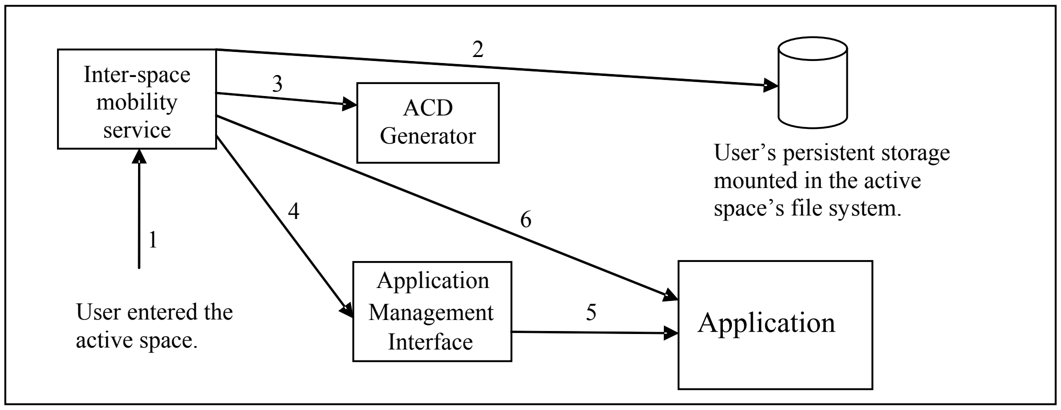

Inter-space mobility service
2
3
ACD Generator
User's persistent storage mounted in the active space's file system.
4
6
1
User entered the active space.
Application Management Interface
5
Application

Figure 27. Resuming an Application.

When a user enters an active space, the Gaia Presence Service detects it and sends a notification to the mobility service (1). The active space mounts the user persistent storage in the space's CFS. Every user has a configuration directory that stores preferences for different spaces, information about suspended applications, and personal information. The mobility service reads the configuration directory and obtains a list of suspended applications (2). For each application, the mobility service requires a valid ACD to resume the application. There are two configurable modes of operation: manual and automatic. In manual mode, the mobility service starts a GUI that provides functionality to manually customize the composition of the application. The GUI uses the application AGD and the SR to find compatible resources, and allows users to drive the ACD generation process. The second operation mode (automatic) implements the following algorithm:

1. Check in the user preferences (mounted storage) if the application was previously suspended in this space. If it was, get the generated ACD and instantiate the application.
2. If the application was not previously suspended, check in the user preferences for a default ACD for the application and for the current space. If there is one available, instantiate the application.
3. If there is no default ACD, contact the ACD Generator (3) with the application AGD and the ACD obtained during the application suspension. The ACD Generator uses the AGD to obtain information about the application components, and uses the ACD to learn about how many components to create of each type. With this information, and using an application generation policy and the SR, the ACD Generator generates a new ACD and instantiates the application.

Note that the presented algorithm is just one possibility. There are different alternatives to obtain the ACD. For example, after step 2, the algorithm could check if the active space has a default ACD for the application. Furthermore, the ACD Generator is likely to generate a number of valid alternatives. The current implementation uses the first one, although it is possible to use AI techniques and give a numeric value to each configuration to rank the obtained ACDs. We believe that the use of policies to configure every aspect of the management interface gives users and developers the freedom to customize the behavior of the active space.

After we obtain a valid ACD, the mobility service uses the application management interface to instantiate the application (5). Finally, the mobility service contacts the application coordinator and invokes the *resumeState* method with the CFS path to the application state saved during the application suspension (6). The coordinator forwards the request to the application model, which reads the state and resumes the internal state of the application.

6.4 Application Adaptation

Active space applications are influenced by external changes that affect the composition of the application at run-time, and therefore applications require support to adapt to these changes dynamically. The type of adaptation we describe here is at the application structural composition level, not at the application functional level. Application functional adaptation (i.e. changing the behavior of the application algorithm) is an important feature that has already been applied to traditional applications[68]. The application framework does not impose any restriction on the implementation of the application components, and therefore, it is possible to leverage existing dynamic functional adaptation techniques.

The requirement for dynamic application composition adaptation does not apply to traditional interactive applications running on desktops due to, at least, three main reasons. First, the usage pattern for interactive desktop applications is different from the one observed in active space applications. Desktop users sit in front of the computer and use the local peripherals to interact with the application. If the users move to a different computer, either they restart the application or they start a remote session (e.g. X-Windows[54], and Windows Terminal

Services[69]); it is not possible to split the application among several devices dynamically. On the other hand, active space applications' users are not bound to a single device; they can move around the space and use any available device; therefore they expect the application to move and duplicate functionality to different devices dynamically.

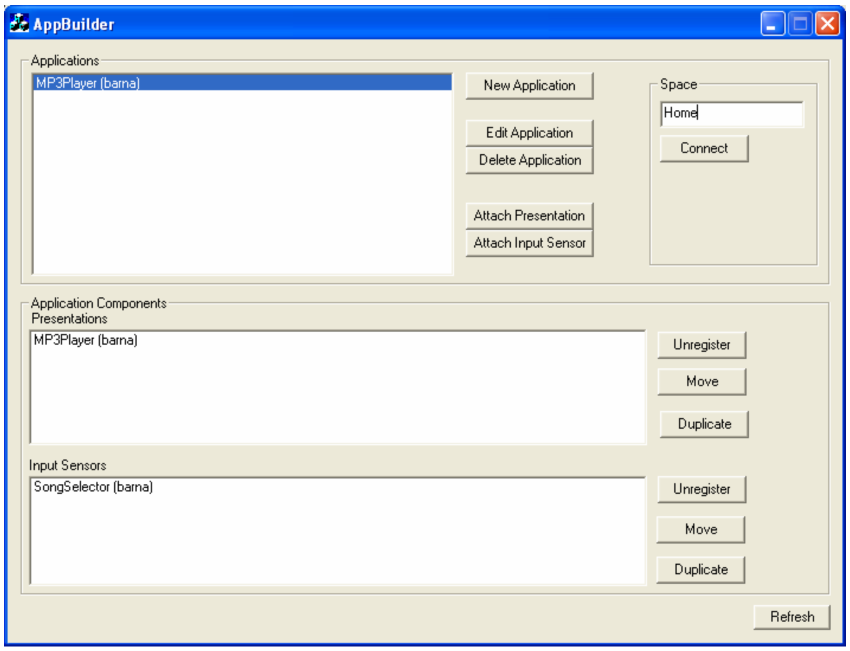
Second, from an abstraction or granularity point of view, the computer defines the execution environment, and therefore, there is no concept or need for splitting the application across different machines. However, in an active space, the active space itself (not the individual devices it contains) defines the execution environment (different abstraction granularities). Therefore, devices contained in the active space become execution nodes of a larger computing abstraction. From this perspective, applications require functionality to alter their composition dynamically to adapt to changes in the active space, and alter the application composition to use the most appropriate execution nodes according to user and context defined preferences.

Finally, most interactive desktop applications are disconnected from external context attributes, and therefore, there is no need to adapt the application composition. The strong connection with context attributes in active spaces requires the application to adapt to new scenarios dynamically.

As an example of structural adaptation, consider a user reading a confidential document on an active office display. When the context of the active space indicates that a user is entering, the application moves the document to the user's personal PDA.

6.4.1 Manual Adaptation

The application framework provides a GUI called Application Builder (Figure 28) that allows users to modify the composition of the application manually according to their preferences.



Applications: MP3Player (barna)

Space: Home

Application Components

Presentations: MP3Player (barna)

Input Sensors: SongSelector (barna)

Figure 28. Application Builder GUI.

The GUI runs in Windows NT and Windows CE devices, and shows a list of application running in the selected space (upper left list). When the user selects an application, the lower half of the GUI displays the presentations and controllers registered with the application. Users can select any presentation and controller, and move it or duplicate it to any compatible device present in the active space. Users can also register new components and unregister existing components with and from the application.

6.4.2 Automatic Adaptation

The application framework uses policies to drive the application adaptation automatically in response to changes in the active space context. Unlike the instantiation mechanism presented in section 6.1, the application framework does not provide any default policies. The reason is that adaptation policies are application dependent. Nevertheless, the application framework provides support for creating adaptation policies easily using the *application bridge* described in section

4.2. Note, however, that it is possible to use the Gaia kernel services to interact directly with the applications and program their adaptation bypassing the bridge component if required.

In order to create a bridge, the policy writer must select an adaptation trigger, a target application, and must provide a script with the adaptation rules. An adaptation trigger can be any application running in an active space, or a *poset* object from the Gaia Context Service. A poset is a service that implements a state machine that models the context of an active space and triggers events when it detects changes in the context. Posets are implemented as applications, and therefore, can be integrated with the bridges directly.

I present next, an example that illustrates how to create and use an adaptation policy. I will use the music player application created in section 6.1 as the target of the adaptation. For the trigger I will use a location application that stores information about the people present in a space. The policy moves the MP3Player from the room speakers to the user's laptop and vice-versa, depending on the number of people present in the space. When the user is alone, the application uses the room speakers, and when someone else enters the room, the application uses the laptop so the user listens to the music with the headset. Figure 29 depicts the two applications with the bridge connecting the location application to the music player application.

Figure 30 illustrates the adaptation script. Each time a user enters or leaves an active space, the location application sends a notification. The bridge receives the notification and calls the script. The script obtains the coordinator from the MP3 Model, and sends a request to the coordinator to obtain the MP3 Player presentation[2]. Next, the script obtains the number of people present in the space from the location application. If the number is larger than one, the policy moves the

[2] The script should get a list of all presentations and filter the list to store only the MP3 Player presentations. I have not added the code to simplify the reading of the script.

presentation to the user's laptop (it uses a library to move the component). If the number of people is one, the script moves the presentation to the room's audio output server.

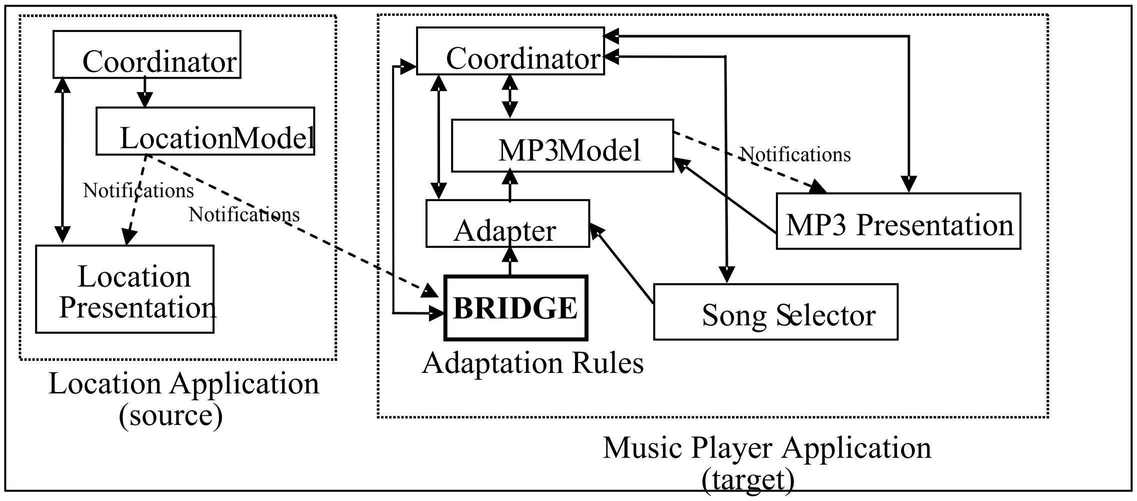


Figure 29. Music Application Adaptation Based on the Number of People in the Space.

```
Function(targetAdapter, sourceModel, sourceEvent)
   player = targetAdapter:getCoordinator():getPresentation("MP3Player")

   if (sourceModel:numberOfPeople()>1) then
         movePresentation(targetAdapter:getCoordinator(),player,"laptop1")
   else
      movePresentation(targetAdapter:getCoordinator(),player,"roomaudiosvr")
   end
end
```

Figure 30. Music Player Adaptation Script.

The bridging mechanism combined with the scripts, simplifies the development of complex adaptation rules.

6.5 Application Fault-Tolerance

When the components of an application are distributed among a number of devices the risk of failure increases (e.g., network errors, device failures, component errors). However the application reaction to failure depends on the type of failure, the nature of the application, and the user and active space preferences. For example, if the play list presentation of a music application crashes, the application may simply continue and update the state of the coordinator, or may decide to restart the component automatically. The default coordinator provided by the application framework implements a fault-tolerance policy that detects when an application component stops functioning and automatically detaches it from the application using the coordinator interface. It is possible to use configurable policies (inheriting from the default coordinator or using scripts) to define more sophisticated fault-tolerance policies. For example, if the coordinator detects that an application component crashes it can contact the execution node, restart the component, and register it with the application. If the execution node is not responding, the policy can find a new appropriate execution node (e.g., using external services) and restart the component there.

Chapter 7

Active Space Applications and Computational Reflection

An interesting property of active space applications is the clear distinction between the semantics of the application and the application interaction configuration (application composition). The number and type of resources present in an active space implies that there are several valid application interaction configurations. However, regardless of the different compositions, the semantics of the application remain constant. For example, consider a slideshow application running in an active space with five displays and three touch sensitive panels. There are 31 possible ways of displaying the slides (i.e. sum of combinations of five elements in groups of one, two, three, four, and five elements), and 7 ways of instantiating the slideshow controllers using the touch sensitive panels. Combining both results we obtain a total of 217 valid application interaction alternatives. However, despite the application composition, the semantics of the application (displaying and controlling slides) does not change. The distinction between the application semantics and the application interaction configuration makes the active space application development process tractable.

The application framework described in this thesis is based on three main properties:

- **Separation of concerns:** The framework clearly separates the application semantics from the application interaction configuration.

- **Self – awareness:** Applications require information about the properties of the current application interaction configuration. This information is essential to allow applications reason about their current configuration.
- **Dynamic Adaptation:** The dynamic behavior of active spaces requires the ability to adapt the application interaction configuration at run-time. Choosing a static configuration during the application instantiation is not enough. The framework allows altering the application interaction configuration at run-time, based on different properties including context changes.

These three properties are the foundation for a type of systems known as Open Implementations [70], which are based on Computational Reflection[71]. The application framework leverages Computational Reflection to manage the complexity involved in the development of active space applications, allowing developers to concentrate on the application domain functionality (semantics) and providing mechanisms to automate the application interaction configuration dynamically.

7.1 Computational Reflection and Open Implementations

The concept of computational reflection was first introduced by Bryan Smith in 1984. It refers to those systems that provide functionality to access (introspect) and modify their own state. Changes in the internal state of the systems are causally connected to the behavior of the system itself. Computational reflection allows customizing the system to different scenarios, properties and environments.

Computational reflective systems are often referred to as Open Implementations. An Open Implementation system distinguishes base-level and meta-level. Base-level implements the application domain functionality, while the meta-level implements the functionality to access and alter the internal structure of the base-level. For example, assume an application that manages personnel data and uses a sort algorithm. The application base-level is the code that manipulates the personnel data and uses the sort algorithm. The meta-level implements the functionality to obtain information about the sorting algorithm, and functionality to replace it, depending on the number and type of personnel records. Changes in the meta-level are transparent to the base-level and affect the behavior of the application (e.g., improved performance).

7.2 Open Implementation Application Framework

The application framework described in this thesis is an Open Implementation. The model, presentation, adapter, and controller, implement the base-level, while the coordinator implements the meta-level.

The implementation of the base-level components is independent of their composition. For example, the functionality implemented by a model component for a music application is independent of the number and type of presentations and controllers registered with the application. Similarly, a power point presentation (renders power point slides) is not affected by other presentations and controllers. Every component encapsulates well defined functionality that remains constant regardless the way application components are composed.

The coordinator provides functionality to control the composition of the application base-level interactive components (presentations, adapters, and controllers). It exports information about what components are registered with the application, and provides functionality to register new components and remove existing ones at run-time. The functionality provided by the coordinator allows configuring the application interaction mechanisms (application meta-level).

A music player application, for example, provides functionality to play music files stored in a play list (base-level). The music can be playing in the active space audio system, a laptop, or a PDA and the play list can be displayed in the user's phone, PDA, or smart watch (meta-level). While the goal of the application never changes (play music), the composition of the application interactive components to achieve the goal does. Application developers should concentrate on the application domain and not in the composition of the application which cannot be known a priori. Computational reflection provides an elegant and effective solution to model the complexity of active space applications.

7.3 Preserving the Application Semantics

The application framework leverages computational reflection to separate the semantics of the application from the application interaction configuration. However, this separation does not address the issue of application semantics preservation. The framework must guarantee that when the composition of application base-level components changes, the semantics (goal) of the application remains unaltered.

I define active space applications as a 4-tuple *A* consisting of a model, an adapter, a set of presentations, and a set of controllers. I also define a function alterInteractiveConfiguration that receives an application App, and returns another application App'. The new application App' has the same model "m" as App, an adapter a' that can have the same mappings as a or can introduce new mappings and remove existing ones, and presentation and controller sets, which can be a subset, a superset, or the same set as the original one (Figure 31)

$A = < \text{m, a}, P, I >$

m: Application Model.
a: Application Controller.
p: Application Presentation
i: Application Controller
$P = \{ p_1, p_2, …., p_n\}$
$I = \{ i_1, i_2, …., i_n\}$

alterInteractiveConfiguration(application) : $A \rightarrow A$

(alterInteractiveConfiguration(App) = App') Λ (App' = $\{ m, a', P', I'\}$)

Figure 31. Preserving Application Semantics While Altering the Interactive Configuration.

The application framework defines the model as the component that implements the logic of the application and exports an interface to manage the application state. The model encapsulates the semantics of the application and is independent of the mechanisms to present its state (presentations) and the tools to alter its state (controllers). Therefore, as long as the model of an application does not change, the semantics of the application remain constant.

The *alterInteractiveConfiguration* method guarantees that the resulting application maintains the same model as the original one regardless of changes in the application interaction configuration. Therefore, it guarantees that application semantics are not affected by meta-level changes.

Chapter 8

Framework Evaluation

The primary goal of the application framework is to assist developers in the implementation of active space applications. This assistance includes shielding developers from the complexity associated to active spaces, providing mechanisms to customize applications to different active spaces automatically, and offering a generic infrastructure to support the development of possibly any type of application the user can devise.

I have organized the framework validation discussion into two issues: application development and performance. The goal of this thesis is to proof that the combination of abstracting active spaces as a computational environment and using computational reflection, effectively simplifies the development of active space applications. Therefore, I place especial emphasis on the application development process and give performance details to show that the response time of the applications is comparable to traditional desktop applications.

8.1 Application Development Process

In order to validate the application framework I present a pseudo-algorithm that illustrates the steps required to build an active space application. The algorithm is generic for any possible type of application, is independent of specific active space instances, and does not require a-priori knowledge of existing resources. Furthermore, any application built with the framework can automatically benefit from the mapping mechanisms and the application management functionality.

The application framework, as a computational reflective infrastructure, separates base-level from meta-level functionality. Base-level is related to the application domain functionality and is independent of additional details such as fault-tolerance, mobility, and adaptability. The base-level is implemented by the Model, Presentation, Adapter, and Controller. The meta-level addresses the issue of application base-level component composition and is implemented by the Coordinator. The application framework management interface interacts with the application meta-level.

Developing an application requires designing and implementing the application base-level. The application meta-level is application domain independent and in most of the cases developers leverage the default implementation provided by the framework. The application base-level implementation process can be summarized as follows:

1. **Application functionality decomposition**. The first step requires developers to separate the application core functionality (model) from the mechanisms to present the application data (presentation), and the mechanisms to interact with the application (controller).
2. **Application model implementation.** The model is the core of the application and it implements the application algorithm. As part of the application model implementation, developers must define an interface to access and alter the state of the model. This interface is independent of specific presentations and controllers. The model updates registered listeners when its state changes. The update includes a "hint" with information about the change in the model. During the design of the model, developers provide a list of hints and their meaning.
3. **Application presentation implementation.** At this step, the application developer decides what aspects of the application must be presented to the user. The presentation

uses the model's interface to obtain application data in response to model updates. The developer uses the information about the model's hints to program the reaction of the presentation.

4. **Application controller implementation.** The controller component mediates interaction between an external agent (e.g., a user, an active space, another application, an adaptive algorithm, and context) and the application. The component uses the model's interface (via the adapter) to alter the state of the application. As part of the controller implementation, developers define the adapter mappings to adapt the controller to the model. If the controller is specifically designed for the application, the adapter simply forwards the requests (the controller invokes the right methods), and therefore developers do not need to define any mappings. However, if the developer reuses an existing controller, he or she must update the adapter to match the controller's requests to the model's interface.
5. **Creating the Application Generic Description.** This description contains information about the model of the application, presentations, and controllers. Developers must specify the resources required by each component as well as their cardinality. The application framework uses this description to generate application customized descriptions.
6. **Providing a description for each component.** Developers must provide an XML description for each component. This description specifies the type of each component as well as a number of configurable properties, which are stored in the space repository. This information is used to enforce type safety (prevents assembling together incompatible components) and allows browsing for components.

The six steps described by the algorithm are generic enough to build any active space application. The eight applications described in chapter 9 are built according to this algorithm. The algorithm emphasizes a number of issues:

- The communication between presentations and the model is not predefined. It is possible to use different mechanisms including RPC, streaming, or even raw sockets.
- The framework guarantees application consistency based on the notification mechanism implemented by the model. All registered listeners (presentations and controllers) are synchronized with the model.
- Building presentations and controllers is optional. Users can look for pre-existing components and use them with the new model. Furthermore, the number of presentations and controllers is not restricted to the components built during the application development. It is possible to create new components later on and use them with the application.
- The adapter simplifies reusing controllers with different applications.
- Finally, during the application design, users do not have to concern about the details of different active spaces. The framework encapsulates these details. Furthermore, mechanisms for application management and specialization are independent of the base-level of the application and therefore users are not required to provide specific functionality.

8.2 Building an Active Space Application without an Infrastructure

In order to stress the benefits of the application framework, I compare the development of an active space application with and without support from an external infrastructure. This external infrastructure refers both to the meta-operating system and the application framework (the framework relies on the meta-operating system).

The discussion is based on a slide show application that provides functionality to present slides in different displays simultaneously, functionality to control the slides from different devices simultaneously using a GUI similar to a VCR interface (i.e., start, stop, next, previous), functionality to obtain the slide show data from any device present in the active space, including devices introduced by the users (e.g., PDAs, and laptops), functionality to move presentations and controllers to different devices, and functionality to suspend and resume the application.

8.2.1 External Infrastructure Support

This section uses the algorithm presented in section 8.1 to describe the steps required to build the application.

1. **Application functionality decomposition.** The application consists of a model that stores information about the current presentation (the name, data source, and slide number), a presentation that renders the current slide (uses Microsoft's Power Point), and a controller that allows users to start and stop the presentation, and move to the next and previous slide.
2. **Application model implementation.** The model stores information about the current presentation and provides an interface to select a presentation, obtain the presentation, start and stop the presentation, move to the next and previous slide, and save and restore the application state (presentation file name and current slide number). The model leverages the context file system to obtain the data from any device (local and remote) including devices introduced by the people present in the space. The model defines four notifications (hints) that are encoded as strings ("!start", "!stop", "!next#<number>",

”!previous#<number”) and are sent when invoking start, stop, next, and previous (in this order) using the model’s default *change* method.

3. **Application presentation implementation.** The presentation interacts with Microsoft’s Power Point using the COM interface. When the presentation gets a “!start” notification, it contacts the model, retrieves the presentation data using the context file system’s handle to the file returned by the model, and renders the first slide of the presentation. When receiving “!next#<number>” and “!previous#<number>” the presentation renders the slide number encoded in the notification (encoding the number prevents the presentation from keeping state). Finally, when the presentation receives a “!stop” message, it removes the slide from the display. Furthermore, the presentation implements the *attachModel* and *detachModel* methods so it automatically displays the slide when attached to a presentation that has already started, and removes the slide from the display when removed from the application (it reuses most of the functionality of the start and stop methods).
4. **Application controller implementation.** The controller is a GUI consisting of four push-buttons: start, stop, next, and previous. Every button calls a method in the model with the same name and therefore no adapter mappings have to be defined.
5. **Creating the Application Generic Description.** The AGD contains an entry for the model, an entry for the presentation, and an entry for the controller. The model entry defines a cardinality of exactly one instance, and requirements that include an execution node. The presentation entry defines a cardinality of minimum one and maximum undefined, and requires a display. Finally, the controller has a cardinality of minimum one and maximum undefined, and requires a touch sensitive device.

6. **Providing a description for each component.** This last step is required so the components can be properly registered with the active space infrastructure, and the coordinator can enforce type safety when composing the components. The description consists of an XML file with information including category, type, subtype, and description.

The application developer concentrates on the application functional domain (managing a slide show presentation). The application framework provides functionality to synchronize all application components, customize the application to different active spaces, move and duplicate presentations and inputs sensors to any compatible device present in the space, suspend and resume the application, add and remove presentations and controllers at run-time, detect crashing components and react automatically (removing them from the application or restarting them), functionality to terminate the application, removing the components from the multiple devices that host their execution, and provides mechanisms to compose the application with other existing applications (application bridges).

Based on our existing presentation manager application, the model requires 350 lines of code, the presentation has 350 lines (excluding the code to interface with the COM object), and the controller (considering the code to interact with the model and excluding the code for the GUI) requires 80 lines (these numbers include function headers, blank separation lines, debug code, and error handling). Therefore, an application developer is able to build an application that fully benefits from the resources present in the active space to manage slide show presentations with 780 lines of code. Furthermore, all the code is related to the functional aspect of the application.

8.2.2 No External Infrastructure Support

This section summarizes the steps required to implement the application without any external support. In order to simplify the discussion, I split the infrastructure support into meta-operating system support, and application framework support. No meta-operating system support implies that the active space is an environment with a large collection of heterogeneous and isolated network-enabled devices. Developers are responsible for any support infrastructure to integrate the devices. On the other hand, no application framework support implies that the developer can leverage the meta-operating system functionality but has no support framework to build applications.

Building the presentation manager without meta-operating system support is a paramount effort (similar to building an application for a PC without OS support). The developer has to implement (at least) mechanisms to detect devices present in the space, mechanisms to allow software components to interoperate over the network (probably leveraging a middleware infrastructure), mechanisms to manage components remotely (e.g., creation and termination), mechanisms to detect components crashing (to react accordingly and preserve the application execution), and mechanisms to access data stored in different devices, including devices introduced by people present in the space. The application developer would be responsible for building all the functionality, which is not related to the application domain functionality. Furthermore, unless the support functionality is built in a generic way, future applications would require re-implementing similar functionality. As a result, it is clear that the support provided by a generic meta-operating system-like infrastructure is essential for developing active space applications.

I will assume now that the developer has support from a meta-operating system but has no application framework support. The application uses multiple devices simultaneously and therefore requires a distributed implementation. In order to display the slides, the application provides a service that receives the slide data and renders the slide on the display. This service runs on every display the user selects to present slides. The instances of this viewer service require an additional service to coordinate them. This new service (coordinating service) provides functionality to create viewers in displays present in the active space (using the meta-operating system), stores a reference to all instantiated viewers, provides an interface to iterate over the slides, and stores information about the current slide number and presentation. When the service receives a request to control the slides, it forwards it to all registered viewers. In order to control the slides, the application developer implements a GUI-based service (VCR controller) that interacts with the coordinating service's interface. This new service can be instantiated in multiple devices simultaneously. The coordinating service provides functionality to instantiate the controller service in compatible devices located in the active space. As defined by the application specification, it must be possible to move and duplicate the slides and controllers to any compatible device present in the space. In order to accommodate this requirement, the developer extends the coordinating service to provide such functionality. The moving functionality creates a new viewer or controller in a new device and terminates the original one. Viewers contact the coordinating service to get the current state, so after being moved they display the current slide. Finally, the coordinating service is extended to save and restore the state of the application. The state includes number and location of viewers and controllers, and current slide number and presentation name. Note however, that this functionality allows restoring the application in the same space, assuming that the resources being used do not change (one of the viewer could be running on a PDA). If the resource configuration changes the coordinating service must provide additional functionality to adapt the application to the new environment.

There are two interesting aspects regarding the customized application. First, the developer is responsible for implementing the application domain functionality as well as the functionality to manage the application (instantiation of viewers and controllers, moving functionality, saving and restoring the application, component assembly, and synchronization of viewers). Second, the resulting application closely resembles the model proposed by the application framework (viewers are presentations, controllers match the Controller component, coordinating service combines functionality from the model and the coordinator).

An application developer that does not use the framework is responsible for implementing all the management functionality for every application. Furthermore, since the management of the application is combined with the application domain functionality, it is hard (if not impossible) to easily tune the management of the application. The application framework's separation of concerns makes it possible to customize the management of the application using policies that are application domain independent. Furthermore, the application framework provides more default meta-level functionality than the one provided by the custom built application (e.g., fault tolerance, application mapping, and context-awareness).

8.3 External Evaluators

By the time of writing this thesis, two additional people (besides the author) have already used the framework to develop applications for active spaces. One person wrote the synchronized PowerPoint application and the inter-space mobility functionality, and the second person wrote a ticker tape, and a location application. In this section, I summarize their experience using the framework, and whether or not they considered the framework to be useful. The section is divided into four subsections, each one of them corresponding to a question that was giving to the developers.

8.3.1 Understanding the Application Partitioning

The goal of the question was to understand how easy or difficult was to get used to the application structure defined by the framework. The developers agreed that the application partitioning defined by the framework was adequate for the type of applications they built, and was also easy to get used to it. One of them mentioned that reusing MVC made it straightforward to understand the partitioning.

8.3.2 Scope of the Application Framework

The application framework is designed to support the construction of possibly any type of active space applications. Therefore, it has to be generic enough to address different types of applications while providing enough functionality to make it useful. According to the two application developers, the framework supported and simplified the development of their applications, and they did not require any changes on their original designs to accommodate their applications to the framework.

8.3.3 Complexity of the Application Framework

The application framework is built on top of the Gaia meta-operating system and uses CORBA as the default communication middleware. Both application developers agreed that learning CORBA was time consuming, and setting up an application required too many steps: creating the XML descriptions, updating the Gaia repository with the IDL files, and setting up the AGD.

Furthermore, one of the developers pointed out that learning Gaia to leverage its functionality required a significant amount of time.

We have already addressed some of the issues. For example, Gaia provides now an API that allows developers to leverage the functionality using a simple C++ interface. This interface hides several details such as contacting the appropriate server and sending CORBA requests. The developer who mentioned the problems with Gaia did not have access to the wrapper. However, the second developer was able to leverage the API, and pointed out that even though he did not know details about Gaia (thanks to the API) he was able to leverage its functionality. Our own experience proves that the wrappers have simplified the utilization of Gaia services.

Regarding the amount of details required to setup an application, we plan to build a "wizard" application that will assist developers with all required steps. We believe such application will effectively simplify the application setup.

Finally, regarding the time required to learn CORBA, it is difficult to give a solution. Active spaces are distributed systems and therefore require a middleware infrastructure to enable distributed object interaction. There are a number of alternatives to CORBA (SOAP[4] and RMI[5]), however, they also require a learning process. As these technologies become more pervasive, they will become common topics in CS classes (they are already taught in most universities), and therefore developers will be familiar with them.

8.3.4 The Framework Allows Exploiting the Resources in the Active Space and Hides the Associated Complexity.

Both developers agreed that the framework was useful to develop applications that use multiple heterogeneous devices simultaneously. The meta-level functionality and the management interface were the two most appreciated aspects of the framework. Both developers worked on the application domain and leveraged the already existing functionality. They pointed out that the ability to move components of the application by default was probably one of the most useful mechanisms because it simplified the coding of their applications. Furthermore, one of the developers mentioned that once the application components are initialized properly, the GUI development becomes a much larger limiting factor in application development than the framework itself. Finally, the same developer mentioned also that he found the application bridge mechanism to be a useful mechanism to easily combine application functionality. The application bridge mechanism was completed after one of the developers left. Therefore, he could not comment on such functionality.

Based on the reviews of the developers, the framework provides useful functionality to develop active-space applications. Most of the problems that arose during the application development process were related to bugs (the framework was still being completed at the time both developers used it), poor documentation of Gaia (although the Gaia wrappers partially solved the problem), complexity of the middleware, and lack of a mechanism to automate the application setup. As we solve some of the problems, and more example applications are built, we expect to simplify event further the application development process..

8.4 Application Performance

The goal of this thesis is to provide support to build active space applications. Therefore, it does not really help to use performance numbers to prove that the application framework is indeed useful. I have evaluated the framework in the previous sections based on whether or not it is sufficient for a large number of applications, and whether or not it helps developers to exploit large collections of resources contained in an active space. Finally, I have included the opinion of two early developers who built a total of three applications that we use regularly.

I agree, however, that regardless how good the support is, if it is an order of magnitude slower than a standard application, the overall framework is doomed to failure. The interactive response times of all applications we have built, are comparable to standard PC applications. For example, in the slideshow application, the time it takes to move to the next or previous slide since we press a button in a PDA, is the same as in a standard Power Point application running in a PC. The time is bounded by the Power Point rendering engine, not by the network-based requests introduced by the framework. We have observed the same results in all the other applications (e.g., music player, PDF Viewer, Movie Player, and Scribble).

In this section, I include some performance numbers that show that the applications built with the framework perform within acceptable interactive margins. I present timings for application instantiation (including different number of application components), and moving a presentation from one display to another.

All the tests were performed in our prototype active space, which has a 1Gb Ethernet network, 15 Pentium IV at 1.2 GHz and 256MB of RAM, and 4 61" Plasma displays.

Figure 32 illustrates the average time required to instantiate the presentation manager application, which consists of a model, a coordinator, a number of presentations (one, two, three, and four, each in a different display), and one or zero controllers. The time was calculated from the time we select the application until the first slideshow slide is displayed in all displays. The average time increases linearly as the number of presentation increases. The time required to start PowerPoint in one machine manually (and start the slideshow) is 1.98 seconds. Starting the presentation manager using one display and one controller takes 3.28 seconds, while the same application without the controller requires 2.025. Therefore, the impact of the application framework is negligible from an interactive point of view. Current instantiation script instantiates all presentation sequentially. Therefore, it waits until a presentation is properly created before creating a new one. It is possible to implement an optimistic instantiation policy that uses "oneway" method invocations (it does not wait for a response) and simply checks at the end whether or not all components were created successfully (interacting with the space repository). In this case, the time would be almost constant, regardless the number of presentations because all presentations are instantiated in parallel.

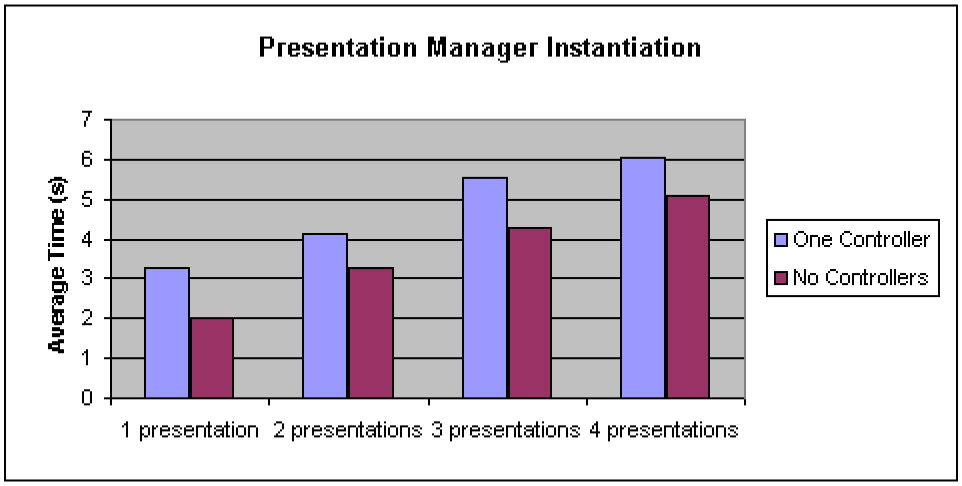


Figure 32. Average Time to Instantiate the Presentation Manager Application.

Finally, Figure 33 illustrates the average time required to move a slide from one display to another. The time includes creating a new presentation, attaching it to the coordinator, and unregistering and terminating the original presentation.

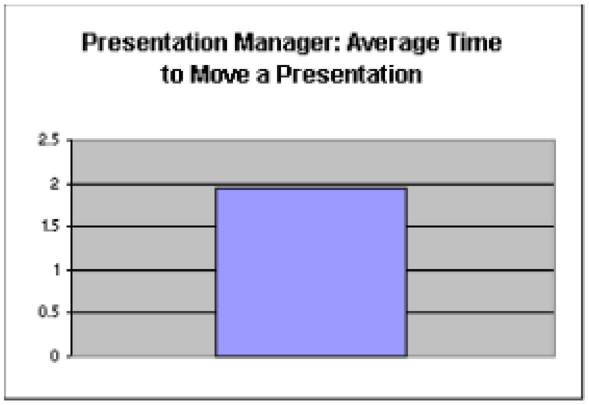


Figure 33. Average Time Required to Move (in Seconds) a Presentation (Slide).

Chapter 9

Example Applications

This chapter describes eight applications based on the framework. Each section presents an application unit and section 9.9 describes aggregated applications we have built using some of the presented applications.

9.1 Music Player

The music player was the first application we created using the framework. It provides functionality to play different music formats (e.g. mp3, wma, and wav), implements a play list controller, provides functionality to synchronize the audio when the player presentation is moved to different machines, and uses the Gaia context file system to aggregate songs from any device contained in the space. This application implements the *suspend* and *resume* methods, and therefore can save and restore its internal state. We used this application as well as the presentation manager to test the session manager described in [62].

9.1.1 Application Components

The music player is composed of six components: Coordinator, Data Exporter, Music Model, Music Player, List Viewer, and VCR Controller. The application reuses the default coordinator and implements the remaining five components. The data exporter, list viewer, and VCR controller are not specific to the music application and they are reused in several other applications.

The *Data Exporter* is a model that simplifies some common tasks related to accessing the Gaia file system. It provides functionality to access a directory and cache all the file names it contains, as well as functionality to open a file – or container according to the file system nomenclature – in different formats using the file system data adaptation mechanisms. It also listens for changes in the file system (new files added or removed) and automatically updates its cache and sends a notification to the listeners. The data exporter provides an interface that allows obtaining the list of files, selecting a file, moving to the next and previous file, and opening the currently selected file.

The *Music Model* inherits from the Data Exporter, and provides functionality to start and stop the currently selected song, go to the next and previous songs, get the current time and move to a specific time, increase, decrease, and obtain the volume, and suspend and resume the application state. Each of the methods implementing this functionality invokes the change method, which automatically notifies the listeners about changes in the application state.

The *Music Player* is a presentation responsible for playing the audio. Current implementation wraps an existing player (Winamp [72]) and interacts with it via the Windows IPC API in response to notifications from the model. This version of the music player does not use streaming. Instead, when the player receives a notification to start playing the audio, it contacts the model, obtains a handle to the music file (i.e. container) using the *getContainer* method from the model, and caches the data in the local device. When the presentation is registered with the application (*attachModel* method), it contacts the model and obtains the current playing time. If the time is not zero, it means that the presentation has been moved and therefore it resumes the music at the stored time.

The *List Viewer* is a software controller that displays a list of entries. When the user selects one of the entries, the list viewer sends a request to the adapter (*setCurrentEntry*) with the name of the selected entry. The adapter forwards the request to the model changing the name of the request according to its internal mapping table (or with the same name if no mapping is defined). For the music player application, the list viewer displays the list of available songs obtained from the model (data cached by the data exporter). When the user selects one of the songs, the list viewer sends a request to the adapter, which simply forwards the request to the model with the same name. The list viewer listens to the model to learn about files added and removed (adds the files to or removes them from the list), and changes in the currently selected entry (changes the highlighted item).

The *VCR Controller* is a software controller with four push buttons: start, stop, previous, and next. Each button triggers a method request to the application adapter with the same name. The application defines two mappings, one for previous (*selectPreviousAvailableEntry*), and another for next (*selectNextAvailableEntry*). The music model defines two methods called start and stop. They match the names of the two events generated by the VCR Controller and therefore no mapping is required.

9.1.2 Application Functionality Details

Figure 34 illustrates an instance of the music player application, consisting of one VCR Controller, one list viewer controller, one music player, an adapter, and a model, and a coordinator. To simplify the diagram, I do not include the coordinator in the figure and will concentrate on the base level functionality. I follow the same approach with all the applications described in this chapter.

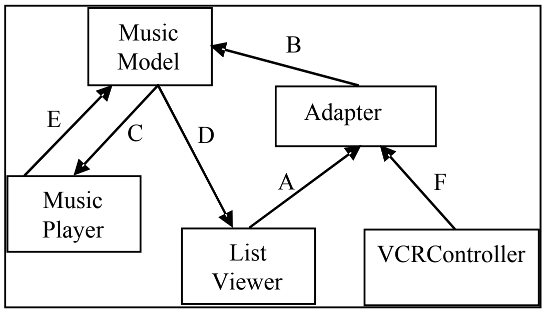


Figure 34. Music Player Functionality Description.

During the instantiation process, the music model receives a parameter with the location of the music files. In most of the cases, we use a context-based path, /type:/mp3, which points to a context-based directory that aggregates files located in different devices. When the list viewer is registered with the application, it contacts the model and requests a list of all available songs. The user selects one of the songs, which triggers a method invocation (*setCurrentEntry*) from the list viewer to the adapter with the name of the selected song (A). The adapter checks its internal mapping list, and since no mapping has been defined, it simply forwards the request to the model (B). The model sets the provided song as "current", and invokes the *change* method with the name of the current song as the hint. The method sends a notification to the listeners (C, D). The music player receives the notification and contacts the model to obtain the file (E). The list viewer gets the name of the current song from the notification and highlights the song in the list. In this case, the list viewer does not require contacting the model. Finally, the user can select actions in the VCR controller, which will trigger similar reactions but with different goals (start and stop the music, and go to the next and previous songs).

The application uses the Gaia file system functionality to detect new songs added to the active space, and existing songs removed from the active space. The music model is registered to the file system updates channel, and when it gets an event, it automatically updates its internal cache, and sends a notification to the listeners. The music player ignores this type of notification, and simply keeps playing the song. The list viewer contacts the model, gets the list of songs and updates the list.

By leveraging the Gaia OS functionality, it is possible to add and remove files to and from the active space dynamically. This allows users to export data contained in their devices. All applications based on the data exporter are aware of data added and removed dynamically.

9.2 Presentation Manager

The presentation manager application provides functionality to create and coordinate slideshows in active spaces. The application reuses Microsoft's PowerPoint and provides additional functionality to display multiple slides in multiple displays simultaneously, control the presentation from different devices, and move and duplicate presentations to different devices contained in the space, including hand held devices.

Our first implementation of the application provided functionality to present a single slideshow file (a power point file) using multiple devices. However, after using the application for some time, we learnt that in an active space with a large number of displays, it would be useful to have functionality to display different slides (from the same or different slideshow file) in different displays. Based on these new requirements we modified our original application to be able to manage synchronized slideshow presentations. Current version of the application is capable of manipulating multiple Power Point files simultaneously and provides functionality to select and synchronize different slides that are rendered in different displays.

9.2.1 Application Components

The presentation manager application is composed of six components: Coordinator, Data Exporter, Presentation Model, Presentation Editor, Power Point Presentation, GIF Viewer, List Viewer, and VCR controller. The application reuses the default coordinator, list viewer and VCR controller, and implements the presentation model, and presentation editor.

The *Presentation Model* inherits from the data exporter model and provides functionality to select a number of Power Point presentations, create and edit synchronization rules for different slides, save and restore synchronization rules, and control an active presentation. The model implements a data structure that stores information about the total number of states implemented by the slideshow, slides associated with each state, and displays used to render each display. In order to guarantee slideshow portability, the model uses the concept of virtual displays. Every Power Point Presentation component (described next) is assigned to a virtual display, which is mapped to a real display at instantiation time, according to the resources present in the target active space. The mapping can be done using the tools provided by the application framework specialization mechanisms, or using the Presentation Editor component (described next).

The *Presentation Editor* is a controller that allows creating the synchronization rules for the slide show, and provides functionality to select a number of existing Power Point presentations, map slides to virtual displays, and map virtual displays to real displays dynamically. The editor updates the state of the model and receives notifications if the state of the model changes. This is useful if the slideshow is created by a number of users working collaboratively.

The *Power Point Presentation* is responsible for rendering Power Point slides. It interacts with Microsoft's Power Point using the COM[73] interface. When the presentation receives a notification to render a slide, it contacts the model to obtain the file (it caches the file locally) and interacts with Power Point to display the requested slide. Because of the synchronization functionality, different presentations display different slides. Therefore, the notification message sent by the model must include an identification field to specify what presentation is supposed to display what slide. The application takes benefit of the *setId* and *getId* methods implemented by the presentation base class. The mapping of virtual displays to real displays triggers the creation of a Power Point presentation component in the specified display. Furthermore, the presentation is assigned an id that matches the name of the virtual display. The model uses the name of the virtual display to indicate what presentation is affected.

The *GIF Viewer* is a presentation that implements functionality to display GIF images. The component is designed to interact with a data exporter model, and uses the provided interface (*getContainer*) to obtain the GIF file. The GIF viewer invokes getContainer and provides a parameter to specify that it wants a container with the data in GIF format. The data exporter leverages the functionality provided by the Gaia file system. It requests the data in the specified format and the file system becomes responsible for transforming the data to GIF, or returning an exception if no valid transformation exists. We use the GIF viewer to display the power point slides in GIF format in hand held devices. The GIF viewer uses the properties of the device where it is instantiated to choose the most appropriate size for the GIF, and whether or not it should be rotated (default behavior in Pocket PC devices).

9.2.2 Application Functionality Details

The presentation manager provides functionality to create and edit synchronized slideshows, as well as to drive the presentation of a previously created slideshow. The creation/edition of a synchronized slideshow is driven by the presentation editor controller. Users select a number of existing Power Point presentations, create a number of states, a number of virtual displays, and then, for each state assign slides to the virtual displays. The presentation editor provides functionality to save the synchronized presentation, as well as functionality to load previously stored presentations. The editor can be used during the presentation to map virtual displays to physical display. However, in most of the cases we leverage the application framework specialization mechanisms.

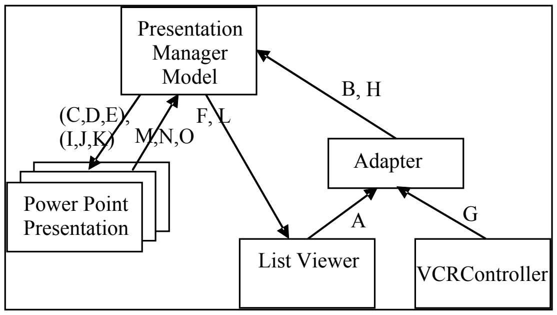


Figure 35. Presentation Manager Functionality Description.

Figure 35 illustrates an instance of the presentation manager. The slideshow has been created previously and therefore the application is used to present the slideshow. The list viewer displays a list of available synchronized presentations, and the VCR Controller provides functionality to control the current slideshow. The user starts by selecting an existing synchronized presentation from the list viewer, which triggers a method invocation (*setCurrentEntry*) with the name of the selected presentation (A). The adapter does not have a mapping for that request name and simply forwards the request to the model (B). The model sends a notification to the listeners (C,D,E,F) and only the list viewer reacts to the model by highlighting the selected presentation. The user selects the "start" button in the VCR Controller, which triggers a method request with the same name (G), which the adapter forwards to the model (H). The model receives the request, traverses its data structure and finds what slides must be rendered by each Power Point Presentation. The model sends a notification with a text parameter including the ids of the affected presentations (I,J,K). The presentations parse the text parameter, and if their id is contained in the text, they contact the model and retrieve the file and the number of the slide they must display (M,N,O).

9.3 Ticker Tape

The ticker tape application provides support for displaying scrolling items sequentially across multiple display devices (Figure 36). The ticker tape serves as an input/output interaction mechanism within an active space. Unlike traditional stock quoting ticker tapes, our ticker tape displays multimedia items, including graphics, and allows assigning specific actions to the scrolling items. Items displayed in the ticker tape can be selected to trigger user defined actions, including launching additional applications, or modifying the state of existing applications. The ticker tape provides a non-intrusive display with information relevant to the space entities and functionality to trigger changes in the active space.

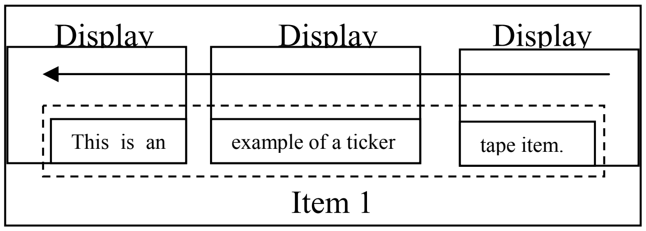


Figure 36. Ticker Tape Item

One main characteristic of the ticker tape is the synchronized and dynamic utilization of multiple display devices. Applications in an active space are not confined to one display device; therefore, a ticker tape item (e.g. text and pictures) displayed in an active space is rendered on multiple devices. When a ticker tape item reaches the edge of one display, it is immediately displayed in the next display. In addition, components in an active space are often mobile, so the ticker tape must be able to respond to devices entering, exiting, and changing location within the active space by attaching, detaching, and re-ordering ticker tape items.

9.3.1 Application Components

The Ticker Tape is composed of four components: Model, Display Controller (DC), Sequencer Controller (SC), and Coordinator. The ticker tape implements the first three components and reuses the default coordinator implementation provided by the application framework.

The *Ticker Tape Model* is responsible for orchestrating the sequential handling of scrolling items across the different displays used by the application. The model associates an index to each scrolling item, and stores an ordered list of ids for each DC, so it can dispatch notifications to the appropriate DC when an item needs to be displayed. It also contains functionality for adding, updating, and removing scrolling items. A scrolling item is stored in the model as a set of attributes, including size, color, font and content of text, the path location and size of pictures, scrolling speed, and other attributes to determine how items are rendered by the DC, and for how long.

The *Ticker Tape Display Controller (DC)* is responsible for displaying scrolling items in a display when the model sends the appropriate notification, and notifying the model when its scrolling item reaches the edge of the display so that the next display controller can be notified. In addition, the DC is responsible for detecting and notifying the model when users select a certain scrolling item so that the model can execute any functionality associated with that item. Upon receiving a notification from the model to display a scrolling item, a DC checks if the notification is intended for it. If so, it requests the set of attributes associated with the item from the model, and renders and scrolls the scrolling item. Pictures and other multimedia entities are not transferred from the model to the DC, instead a file path is used so the DC uses Gaia's context file system [21] to acquire a copy of the entity to be rendered.

The *Ticker Tape Sequencer Controller* is a tool that allows users to change the ordering of the displays used by the ticker tape. It receives the current ordered list of DC from the model and allows a user to enter a new ordering. Currently, the DC can only be sequenced manually, although once more advanced proximity location services are deployed in Gaia it will be possible to automate sequencing based on device location data.

9.3.2 Application Functionality Details

Figure 37 illustrates an instance of the ticker tape application using two DCs and the Ticker Taper Sequence Controller. The user enters the text and the path of the picture to display in the ticker tape. The adapter receives the message, checks for a mapping, and since no mapping has been defined, it simply forwards the request to the Ticker Tape Model (B). The model stores all

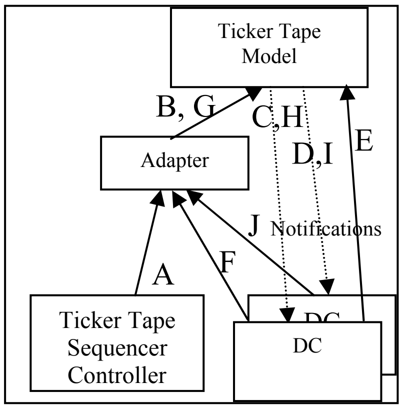


Figure 37. Ticker Tape Functionality Details

the fields for the scroll item and notifies all listeners that a new scroll item is available for display on the first display according to its internal display list. The model sends a notification containing a string with the index number of the new scrolling item and the id of the target DC (C, D). The id assigned to the DC in the forefront matches the one included in the notification, so the DC calls a method on the model to retrieve the scroll item fields (E); the DC uses the Gaia file system to retrieve the image. Next, the DC renders the item using the attributes contained in the item structure and scrolls it across the display. When the scroll item reaches the left side of the display, the DC calls a method on the adapter to notify that the next DC has to begin displaying the item (F). The adapter receives the message and forwards it to the model (G). The Ticker Tape Model notifies all listeners with a message containing the display id of the next DC according to the model's internal display list (H, I). This time, the DC in the background has the correct id, so it calls a method on the model and follows the same steps as the previous DC. The time to send text from one display to another is smaller than 100ms, and therefore negligible from the user point of view.

9.4 Location

The location application provides functionality to track people in some defined geographical region. The application relies on sensor data provided by the underlying infrastructure (Gaia OS) to detect the position of the users. Current implementation of the Gaia location service provides information at room granularity. That is, we can detect whether or not a user is present in a room, but not where in the room the user is located. Once new location services are deployed in Gaia, the location application will be able to give more accurate data about the users' position.

We currently use the application to manage location of users in the DCL building. The main novelty of this application is that it has components running in multiple active spaces simultaneously. In the case of the DCL, we have created a hierarchy of active spaces, where the DCL is the root (a.k.a. *domain space*), and each active office and lab are child nodes. The current tree has only two levels, although it is possible to define more complex hierarchies (i.e. DCL→First Floor→(1110,1120)). The emphasis of the section will be on the application functionality and not in the active space hierarchy, which is part of the future work in Gaia.

9.4.1 Application Components

The location application implements three components, Location Model, Location Presentation, and Location Controller, and reuses the default coordinator.

The *Location Model* provides functionality to store and update information about users and their locations and provides an interface to query about user location. The model stores information about the user name, the name of the space where he or she is located, and the date and time the user entered and left the space.

The *Location Presentation* is a graphical presentation that displays information about user location. Users can select a user name and get updated information about his or her position, or can select a space and learn about present users.

The *Location Controller* registers with the person discovery channel to learn about users entering and leaving the space. When a user enters or leaves, the location controller sends an event to the model via the adapter. There is one instance of the controller for each active space.

9.4.2 Application Functionality Details

This section uses the DCL active space hierarchy to illustrate how the location application works. This hierarchy consists of a domain active space (DCL), room 2401, and room 3231. Figure 38 depicts the DCL active space hierarchy (left) and the application instance (right). When a user enters or leaves a space, the location controller sends a method request to the adapter (A), which forwards it to the location model (B). The location model updates its internal state and notifies all presentations (C, D).

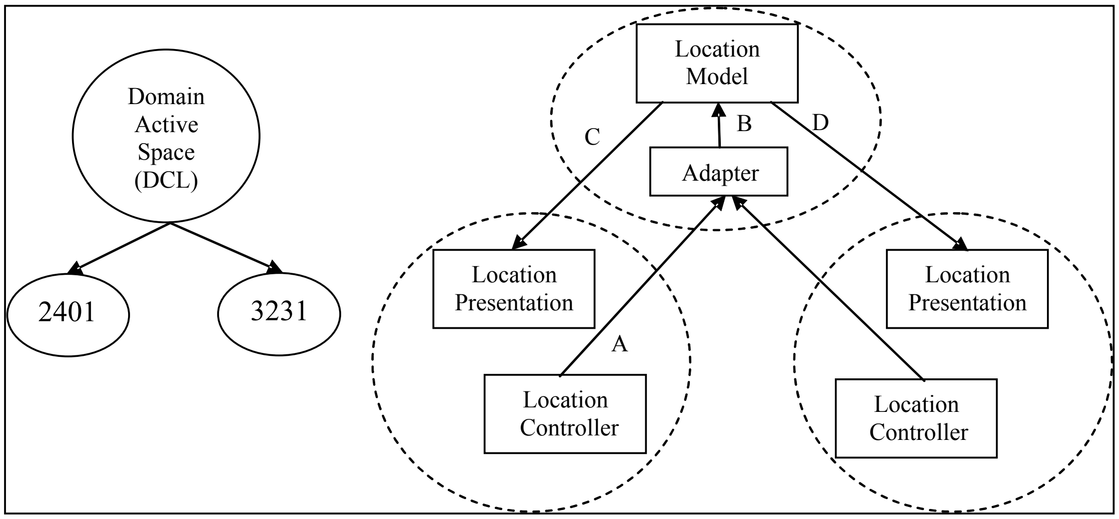


Figure 38. DCL Active Space Hierarchy(left) and Corresponding Location Application Instance (right).

The location controller relies on the functionality provided by the presence service to detect people entering the active space. The presence service uses an event channel to broadcast information about users entering and leaving an active space; the controller registers to the channel and forwards the event to the model.

9.5 PDF Viewer

The PDF Viewer application provides functionality to display PDF documents in active spaces. A document can be displayed in multiple displays simultaneously and can be controlled from a number of devices, including touch screens and hand held devices. Users can start, stop, and move to different pages and the application maintains all the graphical presentations synchronized.

9.5.1 Application Components

The PDF Viewer consists of six components: Coordinator, Data Exporter, PDF Model, PDF Viewer, List Viewer, and VCR Controller. The application implements the model and the PDF viewer and reuses the rest of already existing components.

The *PDF Model* inherits from the data exporter, stores the current page being displayed, and provides an interface to move to a specific page and return the current page.

The *PDF Viewer* reuses the Adobe Acrobat application to render the document and to move to user selected pages. The behavior is similar to the Music Player and the Power Point Presentation. When it receives a notification to display the document, it contacts the model and caches the file in the local device.

9.5.2 Application Functionality Description

The PDF Viewer application is quite simple and benefits from the features offered by the application framework. Figure 39 illustrates an instance of the application and the different steps involved in selecting a document and manipulating it. The list viewer displays a list of available PDF documents, and the VCR controller provides functionality to move to different sections of the document. The application defines four mappings to match the events from the VCR controller to the PDF Model interface. The “start” event is mapped to “open”, “next” to “nextPage”, “previous” to “previousPage”, and “stop” to “closeDocument”.

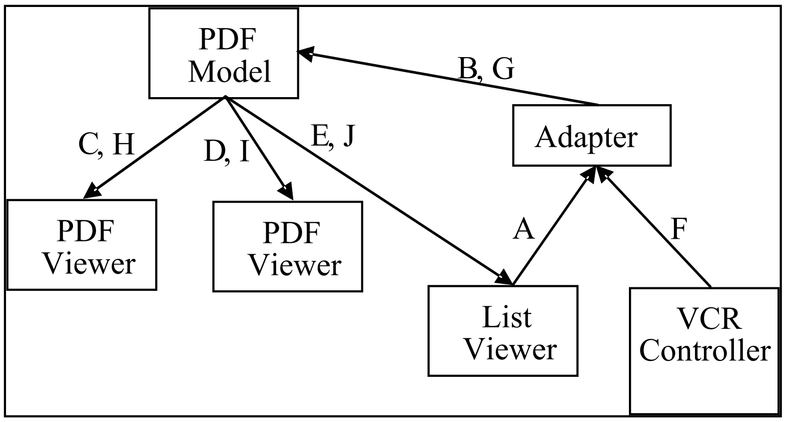


Figure 39. PDF Viewer Application Functionality Description.

The user starts by selecting a document from the list viewer, which triggers a method invocation (*setCurrentEntry*) with the name of the selected document (A). The adapter does not have a mapping for that request name and simply forwards the request to the model (B). The model sends a notification to the listeners (C, D, E) and only the list viewer reacts to the model by highlighting the selected presentation. The user selects the "start" button in the VCR controller, which sends a method request to the adapter (F). The adapter checks the mappings and sends an "open" method request to the model (G). The model receives the request and sends a notification to all listeners (H, I, J). The PDF viewers receive the notification, obtain the file from the model and cache it locally. Next, they interact with the Adobe Acrobat and display the first page of the document. The list viewer component discards the message.

When users select next, previous, and stop, the same steps are followed, and the PDF viewers react accordingly.

9.6 Calendar

The calendar application implements functionality to store appointments, schedule user-defined actions, and set the context of the room to a specific value. For example, users can register an entry in the calendar to launch an application at a specific time and set the context to a value that is appropriate for the application. We use this functionality in our ubiquitous computing seminar to trigger the PDF viewer and the attendance application, and to set the context to "ubi-seminar".

Setting the context allows us to find the relevant PDF documents automatically – the documents are tagged with “ubi-seminar” and the week when they will be discussed.

9.6.1 Application Components

The calendar is composed of five components: Coordinator, Calendar Model, and Yearly, Monthly, and Daily controllers. We reuse the coordinator and implement the rest of the components.

The *Calendar Model* implements an interface to enter new appointments (including recurrent appointments), delete, and edit existing appointments. The model stores the data persistently in an XML file, and implements a delta list to trigger the actions associated with the appointments. The model stores information about the currently selected date, which is modified by the controllers described next.

The *Yearly Controller* is a GUI that displays the twelve months of the year and the days for each year. The days with scheduled appointments are displayed in bold font. When the controller is attached to the model, it retrieves a list of appointments for each day of the current year; then, it keeps this data synchronized using further model’s notifications. Users can select any of the days, which triggers a method invocation that sets the current model’s date to the date selected by the user.

The *Monthly Controller* is similar to the yearly component. The only difference is that it only displays the days for the currently selected month. It also displays the days with appointments in bold, and allows users to select days of the month.

The *Daily Controller* displays information about the appointments for the currently selected day. This information includes a list of appointments, and for each appointment, the starting and ending time, a description of the appointment, location, context associated, action it triggers, and recurrence information associated with the appointment. Furthermore, the controller allows users to edit existing appointments, as well as entering new ones. When the user selects a month, year, or day using the two previous controllers, the model sends a notification with the new selected date. The daily controller automatically presents the appointments for the selected date.

9.6.2 Application Functionality Description

Users interact with the yearly and monthly controllers to select a specific day. For this example we will assume that the user interacts with the yearly controller to check the appointments for a specific day. The user selects a day (A) and the yearly controller sends a method request to the model (via the adapter (B)) to set the current date. The model sends a notification to all listeners (C, D, E). The yearly and monthly components highlight the selected day, and the daily controller presents the appointments for the selected day. Now the user enters a new appointment using the daily controller (F). This component sends a request to the model via the adapter (G). The model stores the information, saves the data in the XML file, and adds an entry in the scheduler for the appointment. Next, the model sends a notification to all the listeners (H, I, J). The yearly and monthly components contact the model, retrieve the new appointment and change the affected day to bold (the monthly controller only implements this functionality if the current month matches the month of the new appointment).

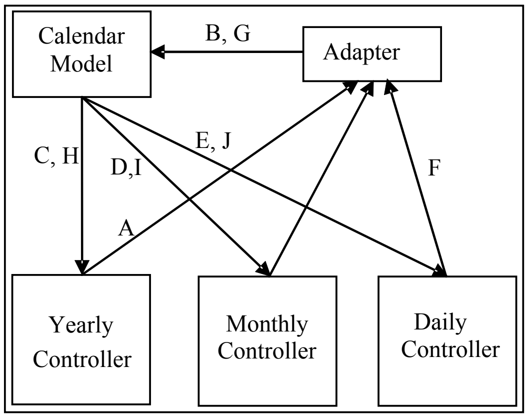


Figure 40. Calendar Application Functionality Description.

Current calendar implementation assumes that the actions stored with the appointments are lua scripts (ACDs, or user-specific code). The model automatically executes these scripts when the appointment is fired.

9.7 Attendance Manager

The attendance manager application provides functionality to store persistent information about users present in an active space. This application is useful to keep track of who was present in an active space on a specific date. We currently use the application to store the name of the people who attended the ubiquitous computing seminar. The application interacts with the Gaia presence service to learn about users entering the active space.

9.7.1 Application Components

This application is composed of three components: Coordinator, Attendance Model, and Attendance Viewer Presentation. As usual, we reuse the default coordinator and implement the remaining components.

The *Attendance Model* provides functionality to store the name of a user and the date when he or she was present in the space, and functionality to retrieve the list of users present in the space on a specific date. This component stores the information in an XML file.

The *Attendance Viewer* presentation implements a GUI with a list box and an entry text field. The list box displays the names of the people present in the space on the date entered in the text field (the component interacts with the model to get the data). The attendance viewer receives a notification from the model each time a user enters the active space.

9.7.2 Application Functionality Description

The functionality of this application is fairly simple. The model listens for events from the person presence channel created by the Gaia presence service. When it receives an event (A), it stores the name of the user and the date, and sends a notification to the attendance viewer (B), which in turn contacts the model, gets the list of users for the date specified in the text entry field, and displays the data (C).

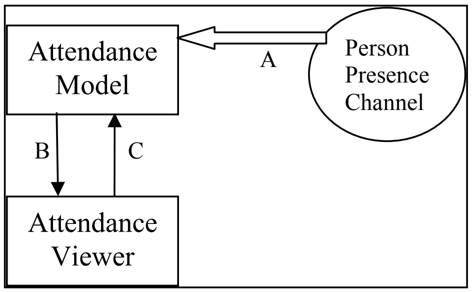


Figure 41. Attendance Application Functionality Description.

9.8 x10 Appliance Manager

x10 is a protocol that allows controlling appliances by sending commands over the electrical network. The appliance manager application provides functionality to manipulate appliances contained in an active space. This application is mostly used in combination with other applications.

9.8.1 Application Components

The x10 appliance manager is composed of four components: Coordinator, x10 Model, x10 Presentation, and x10 Controller.

The *x10 Model* stores information about x10 appliances present in a room and exports an interface to manipulate them. The information includes the control code and description of the appliances. The model stores the information persistently in an XML file, which is stored using the active space context file system.

The *x10 Presentation* implements the x10 protocol and transmits the requests it receives from model to the x10 serial interface.

The *x10 Controller* allows users to manipulate the appliances present in the space. It uses the notifications from the model to keep the state of the different devices up-to-date.

9.8.2 Application Functionality Description

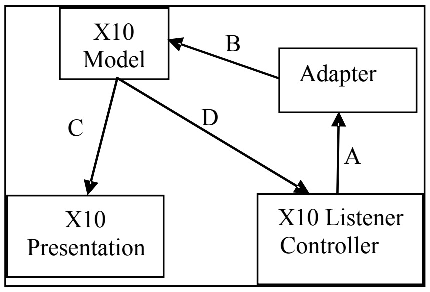


Figure 42. x10 Application Functionality Description.

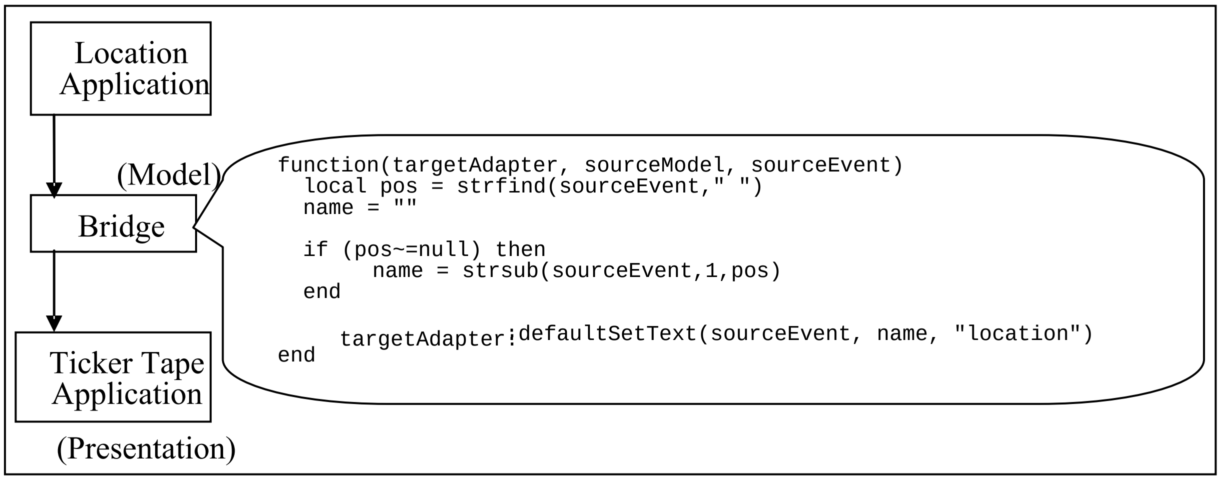
Figure 42 illustrates the functionality of the x10 Application. The x10 controller sends a request to change the status of an appliance to the adapter (A). The adapter forwards the request to the model (B), which sends a notification to the listeners (C, D). The x10 presentation sends a command to the serial interface, and the controller updates the presented information.

9.9 Compound Applications

This section presents two examples of compound applications that use the application bridge mechanism described in chapter 4. One of the goals of this section is to show how the framework allows reusing existing applications unmodified to build new applications. The compound applications presented in this section are based on the previously described applications. None of those applications required any change.

9.9.1 Ticker Tape Displaying Users' Location

The compound application uses the ticker tape application to display location information about users. Figure 43 illustrates the compound application and the bridge script that defines the composition rules. The ticker tape application is registered as a presentation of the location application.



```
function(targetAdapter, sourceModel, sourceEvent)
  local pos = strfind(sourceEvent," ")
  name = ""

  if (pos~=null) then
       name = strsub(sourceEvent,1,pos)
  end

     targetAdapter:defaultSetText(sourceEvent, name, "location")
end
```

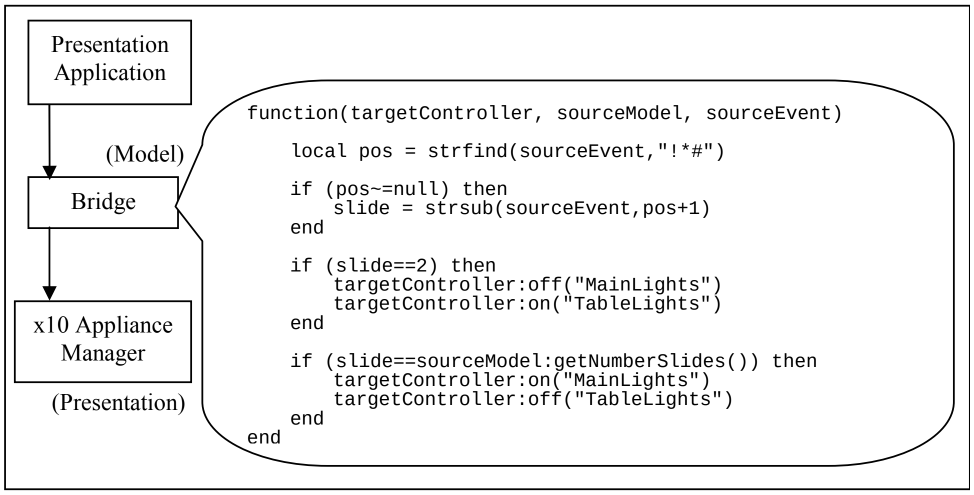
Figure 43. Ticker Tape and Location Compound Application and Application Bridge Script.

When the user enters the active space, the controller of the location application calls a method on the model to report the new user (Andrew) entering active space 2401. The Location Model updates its data structures to reflect the new location report and notifies all of its listeners with the message "andrew has entered 2401". The Location-to-Ticker Tape bridge parses the username, "Andrew" from the message (lines 2 to 7 in the script) and calls a method to create a new scroll item in the tickertape with the text "Andrew has entered 2401" and a picture "users/andrew/andrew.jpg" (line 9). The controller receives the message and forwards the request to the Ticker Tape

9.9.2 Presentation Manager Controlling x10 Lamps

This compound application controls the lights present in the space based on the status of a slideshow. The application bridge switches off the main lights and turns on the table lights when the user moves to the second slide (slide after the title) and switches the main lights on when the user moves to the last slide of the presentation.

Figure 44 depicts the compound application. The Presentation Manager acts as the compound application model and the x10 Application is registered as a presentation. The figure includes the code for the bridge.



```
function(targetController, sourceModel, sourceEvent)

    local pos = strfind(sourceEvent,"!*#")

    if (pos~=null) then
        slide = strsub(sourceEvent,pos+1)
    end

    if (slide==2) then
        targetController:off("MainLights")
        targetController:on("TableLights")
    end

    if (slide==sourceModel:getNumberSlides()) then
        targetController:on("MainLights")
        targetController:off("TableLights")
    end
end
```

Figure 44. Presentation and x10 Appliance Manager Compound Application and Application Bridge Script.

Chapter 10

Related Work

10.1 Active Spaces

We present in this section a number of research projects related to Gaia. I briefly describe each project and compare it to Gaia.

10.1.1 Microsoft Easy Living

The Microsoft Easy Living project [74] focuses on home and work environments and states that computing must be as natural as lighting. The main properties of their environments are self-awareness, casual access, and extensibility. The infrastructure allows interfaces to move, according to user location. The project uses computer vision to recognize gestures and users, and to detect user location. The system uses this information to customize the room accordingly. We differ in that we fundamentally change the way in which applications are built, moving away from the desktop paradigm, and allowing application partitioning on different devices. Furthermore, EasyLiving uses vision techniques to make the room intelligent, so it can automatically assist users. Gaia provides functionality to program the active space. Users can now easily adapt the space to their requirements. Using the same interface we could use vision techniques as a context input and develop mechanisms to automatically adapt the space.

10.1.2 i-Land and Roomware

The i-Land [75] and Roomware [76] research projects present an infrastructure to implement the next generation of human computer interaction. The projects introduce the term cooperative buildings, which refers to the combination of the physical properties of a building and a digital infrastructure that enhances it and allows for new ways of collaboration and interaction. The goal is to offer an environment where it is easy to exchange ideas, digitally record the results of the meetings, search in knowledge bases, and provide tools for group collaboration focusing on multimedia data exchange. These two projects are interested in the interaction with physical spaces (mostly meeting rooms) and collaborative work groups. Our work is similar in that we believe there is a need for a supporting infrastructure. However, we focus on generic spaces (e.g. office and house) which may or may not imply collaborative work. We consider that while some active spaces define a collaborative environment (e.g. meeting room and classroom), other active spaces are mostly single-user based (e.g. office and car). Furthermore, Gaia defines the notion of mobile users that can move their applications and data across different active spaces.

10.1.3 Stanford's Interactive Workspaces

The Interactive Wokspaces [77-79] from Stanford presents an augmented meeting room that promotes group work. The room contains wall-sized touch screens, several projectors, arrays of microphones, speakers, laptops, and PDAs. The project identifies the importance of a high level operating system to coordinate the entities contained in the room. The main components of the software infrastructure (iROS) are: Data Heap (facilitates data movement using a local environment storage mechanism), iCrafter (service advertisement and invocation with an automatic interface generator), and Event Heap (dynamic application coordination and underlying communication infrastructure). Gaia and iWorkspaces agree on the importance of a meta-operating system to manage the physical space and the resources it contains. In fact, there are several similarities in the infrastructure, although the implementations differ significantly at some points.

The Data Heap is equivalent to the Gaia Context File System, however the CFS is implemented as a collection of distributed servers instead of a centralized server. CFS allows tagging data with context information, and provides mechanisms for transformation. CFS makes it possible to dynamically mount and unmount files from devices contained in the space, as well as devices dynamically introduced in the spaces.

10.1.4 CMU's Aura

Aura [64, 80, 81] shares several common design goals with Gaia. Aura emphasizes the notion of mobile users moving around different environments. Their definition of environment is similar to our notion of Active Space. Aura uses the term *task* to identify applications associated with users capable of migrating from one environment to another. Aura defines a software infrastructure to support the execution of these tasks, which maximizes the use of available resources, and minimizes user distraction. The main components of Aura are: Coda (supports nomadic, disconnectable, and bandwidth-adaptive file access), Odyssey (supports resource monitoring and application-aware adaptation), Spectra (adaptive remote execution mechanism), and Prism (captures and manages user intent).

The main difference between Gaia and Aura is that Gaia emphasizes the notion of space programmability. Gaia provides mechanisms to allow users configure their applications to benefit from the resources contained in their current space. Users can interact with multiple devices simultaneously, can reconfigure applications dynamically, can suspend and resume groups of applications, and can program the behavior of applications based on context attributes. Gaia emphasizes the interaction between users and active spaces.

10.1.5 Washington's one.world

one.world's [45] vision is that of a giant distributed system composed of tens of thousands of people, devices, and services coming and going. Their claim is that for this scenario, traditional client-server, peer-to-peer systems are not appropriate. one.world builds on three principles: exposing change, dynamic composition, data and functionality separation. There are also three basic abstractions called environment, tuple, and component. The environment provides structure and control and act as containers for tuples, components, and nested environments. A tuple is a record with self-describing named fields. A component implements functionality and imports and exports statically defined event handlers. The main motivation behind one.world is to expose changes to application developers to cope with the heterogeneous and dynamic nature of ubiquitous computing environments. The software infrastructure provides four basic services: check-pointing (stores the execution state of an environment tree), migration, discovery (allows sending events to services whose location is unknown), remote event passing (communication in one.world is based on asynchronous events), and replication.

Gaia is different in that it is closer to a cluster-based approach. Although ubiquitous computing is a large distributed system, one of the main aspects is that it will become part of the users' physical environment. Users do not live in a gigantic virtual environment. We inhabit spaces where we perform different tasks. From this perspective, Gaia augments these spaces with a software infrastructure. Gaia believes in the interaction of active spaces, which are be hierarchically related. Each active space is a computational cluster, where it is still possible to use RPC mechanisms with some adaptations. Inter-active space communication will be required and although Gaia has not approached that issue yet, it might be possible that an asynchronous mechanism between clusters will be more appropriate.

10.2 Device Adaptation

There a number of projects study how to adapt applications to different types of devices automatically. The goal is to implement the application once and reuse it with a number of different devices. Most of these projects concentrate on user interface adaptation issues and how to connect them to the backend services. Furthermore, all the projects map applications to a single device. They do not address multi-device configurations.

Although the Gaia Application Framework does not provide support for automatic interface generation, it is possible to leverage the functionality of existing approaches to automatically generate presentations and controllers that can be attached to the application.

The application adaptation mechanisms inspired the application specialization mechanisms implemented by the Gaia application framework (ACDs and AGDs).

Roman et al [82] present a model to describe services functionality and interface using an XML description. Services advertise themselves by periodically sending their descriptions. Interested clients use the XML description to generate the GUI and to contact the remote service via an appropriate RPC mechanism (as described in the service description). Eisenstein et al [83] also propose a model for automatic GUI generation based on XML. The difference with the previous approach is that they do not include service invocation information with the GUI description. However, their GUI definition model is more advanced. They allow creating a hierarchical relationship of widgets including relationships among the widgets.

The IBM's PIMA [84] project proposes a model for building platform independent applications. Developers define an abstract application that is automatically customized at run-time to particular devices. The application is composed of a number of tasks connected by connectors, which define the actions to take when going from one task to another. PIMA is based on a runtime that adapts the abstract application description to a particular device. The resulting application is then downloaded and used in the target device.

10.3 Application Frameworks and User Interfaces

This section describes a number of application frameworks, starting with traditional frameworks (MVC, PAC, and ALC) and continuing with new frameworks that address some of the issues related to ubiquitous computing. The section includes also a description about user interfaces that go beyond the graphical approach.

10.3.1 MVC, PAC, and ALV

MVC [23] was designed by Xerox Park as a framework to build applications in their Smalltalk environment. MVC is the foundation of all windowing based systems, commercially implemented by Macintosh and later on popularized by Microsoft. MVC provides a framework to build user interfaces, and divides application functionality into three elements: Model, View, and Controller. The Model implements the logic of the application, stores the state, and provides an interface to interact with the application functionality. The View is a graphical representation of some aspect of the Model. Finally, the Controller is the mediator between a View, an Input Sensor (e.g., mouse and keyboard) and the model's interface. Every View has an associated Controller that defines the interaction rules with the Model.

The presented framework's presentation component is a generalization of the View. It models any output representation, not only graphical.

The Presentation-Abstraction-Controller[85] (PAC) is a framework that specifies interactive application components and their interrelation rules. The Presentation defines the concrete syntax of the application (i.e., input and output behavior of application), the Abstraction corresponds to the semantics of the application (i.e., functions that the application is able to perform), and the Control maintains the consistency between abstractions and presentations. PAC combines the input and output mechanisms in the Presentation component, while MVC requires two components, namely View and Controller. In PAC, Presentations do not need to know the details about the Abstraction. This functionality is encapsulated in the Control, which keeps Presentations and Abstractions synchronized. The advantage is that in PAC, all control functionality is encapsulated in the Control component, while in MVC, the functionality is shared among MVC.

The Abstraction-Link-View[86] (ALV) is also a framework to build interactive applications that are used by multiple users simultaneously. Its goal is to maximize the separation between the user interface and the application logic. The main rationale behind ALV is to foster human-to-human communication and share common data during the interaction to facilitate the interaction. ALV is based on constraints, which allows registering a function with a specific variable. Shall the variable change, the function is automatically invoked. Constraints allow for fine grained control over the synchronization rules, which contrasts with MVC, where the View is responsible for determining what changed in the Model. The Abstraction implements the semantics of the application, the View presents the information managed by the Abstraction to the user and coordinates user input, and the Link stores all constraints and implements the functionality for synchronizing Views and the Abstraction. Every application has at least one

View per user. The Link allows the View and the Abstraction to ignore each other, which simplifies application development and encourages component reuse.

The Active Space Application Framework presented in this thesis, although reusing the original concepts from MVC, uses techniques present in PAC and ALV. For example, the View-Controller tuple defined by MVC does not exist in the presented framework, which explicitly distinguishes between input (Controller) and output (Presentation). A presentation cannot be used to interact with the application. In some cases, the presentation is not even graphical (e.g., audible) which would make interaction impossible. The Controller is responsible for interaction with the application, and can be implemented as a graphical (e.g., array of push buttons) and non-graphical (e.g., temperature sensor) element. The graphical implementation resembles the PAC Presentation and the ALV View, in that it combines both input and output (output only if the Controller is synchronized with the model to update the information it presents) functionality. The framework's adapter is an interface matching component that helps to reuse controllers with different models. However, the controller differs from the PAC Presentation and the ALV View in that it also implements control functionality. That is, when it receives an event from the model it implements functionality to access the model state and update itself. Both the controller and the presentation must know the meaning of the events received from the model, and must know its interface. A future version of the presented application framework could solve this static behavior by using a notification adapter between the model and the presentations and controllers. The notification adapter would be similar to the currently existing controller adapter. It would allow mapping events from the model into method calls in the presentations and controllers, and would map requests from these two components into interface methods implemented by the model. The benefit of the notification adapter is that it can be defined dynamically.

The framework is also similar to ALV in that different users can attach different presentations and controllers to a shared application.

The main differences between MVC, PAC, and the presented framework are that the latter provides a distributed implementation so every component can run in a different machine. As a result, the framework provides a new component called coordinator to explicitly manage the composition of the application (application meta-level). Furthermore, a major difference between the presented framework and the previous frameworks, is that the former provides a specialization mechanism and a management interface (i.e., instantiation, mobility, suspend/resume, adaptation, and fault-tolerance), customized to active space environments.

Finally, the presented framework generalizes the controller time-sharing model defined by the previous frameworks into a space-time-sharing model. For example, according to MVC, all applications' views and controllers share the same input sensors (e.g., mouse and keyboard) and therefore the input sensors must be scheduled. Graspable interfaces [87] introduce the concept of space-sharing, where different input sensors are assigned to different functional aspects of the application, therefore avoiding the need for scheduling them. We combine both approaches into space-time-sharing to model the type of applications we consider. For example, a music application running in an active space uses a PDA to control the current song, and speech recognition to control the sound level (space-sharing), however the same space may host a calendar application that uses the PDA to browse appointments, and speech recognition to control the calendar's functionality (time-sharing). Space-time-sharing allows more than one controller and presentation to be active at the same time, which contrasts with MVC where only one controller-view pair can be active at anytime.

10.3.2 BEACH

BEACH [75, 76] is a component-based software infrastructure that provides support for constructing collaborative applications for active meeting rooms. BEACH applications are similar to the applications we propose in that they contemplate one user exploiting multiple devices at the same time, dynamic reconfigurations, integration of the physical space, interoperation among all resources contained in the space, and they rely on a software infrastructure to access resources contained in the space. However, the main differences between BEACH and our approach are that BEACH concentrates on collaborative applications while we consider both collaborative and single user applications, BEACH is customized for meeting room-like environments while our framework can be used in different scenarios. Finally our framework focuses on user-centrism and mobility, which allows us to move applications to different active spaces. BEACH does not directly address this issue because the target a single scenario. An interesting aspect of BEACH is the definition of reusable classes to simplify the construction of collaborative applications based on multimedia documents. This concept of class-libraries customized for particular environments is something we would like to incorporate in the future. BEACH addresses user interfaces based on graphical elements. However, the application framework described in this thesis provides functionality to create non-graphical user interfaces.

10.3.3 iCrafter

iCrafter [79] is one of the building blocks of iRoom and provides functionality for service advertisement and invocation, and an automatic interface generator. The functionality provided by iCrafter is implemented by different services in Gaia, except the automatic interface generation functionality that is not provided in Gaia. Service advertisement is implemented in

Gaia by the Presence Service and the Space Repository and service invocation is built on top of middleware services (e.g., CORBA).

iCrafter does not define an application as an abstraction. Instead, it assumes that an application is composed of a number of services that belong to the space, and provides functionality to generate a user interface to access them. However, iCrafter does not provide functionality to manage applications, which is one of the aspects of the Gaia application framework. Furthermore, iCrafter does not provide functionality for partitioning applications into multiple devices dynamically, does not have explicit knowledge about the different components that compose the application (presentations and controllers), and does not provide functionality to suspend, resume, or move applications.

The functionality provided by iCrafter to generate a user interface for a service or a group of services allows users to interact with the application. This functionality is equivalent to registering a presentation and/or controller to a Gaia application.

10.3.4 QCompiler

The QCompiler[46, 51, 88] is a programming framework that assists in the development of multimedia applications for ubiquitous computing environments. The framework provides a high-level application specification independent of specific environments, a meta-data compiler that validates the application specification, a binding mechanism to map application descriptions to specific environments, and a run-time meta-data execution (Gaia OS Kernel).

The high-level application specification includes a DAG with the tasks that compose the application (a task is a component or a collection of components), a collection of QoS templates defining different levels of quality, and finally a collection of adaptation rules.

The meta-data compiler translates the high-level application specification into environment-independent and environment-dependent lower-level meta-data representations. The environment-independent representation is a QoS application descriptor independent of specific environments (similar to an AGD), while the environment-dependent representation maps the previous independent representation to a specific ubiquitous computing environment (similar to an ACD). The translation process is responsible for: determining the correctness of each setup configuration, associating an end-user node in each setup-configuration according to the QoS level template, checking quality-aware consistency checks, associating every valid configuration with possible configuration(s) of generic QoS-enabling services, translating the adaptation rules into XML format.

The binding process transforms the tasks in the generated environment-dependent descriptor into executables ready to be deployed (the binding process modifies the original components' code and adds instrumentation code).

Finally the run-time meta-data execution deploys, manages, and controls the application in a specific environment. The QCompiler uses the Gaia OS Kernel as the default middleware infrastructure.

QCompiler and the Gaia Application Framework are similar in that they provide an application model and mechanisms to simplify the development and deployment of ubiquitous portable applications. QCompiler targets multimedia applications while the Gaia Application Framework targets interactive active-space applications. The Gaia Application Framework does not provide any support for multimedia applications, while QCompiler does not provide support for some of the specific requirements of Active Space interactive applications (e.g., multi-device user interfaces and dynamic application composition). Therefore, both frameworks complement each other. As part of future work we plan to study how to combine both frameworks to provide a unified solution that supports both active-space interactive and multimedia applications.

10.3.5 Pebbles

The Pebbles [22] [89] project is investigating partitioning user interfaces among a collection of devices. Pebbles is mostly concerned with issues related to GUIs, and the infrastructure does not provide functionality for dynamically altering the partitioning layout. Our application model focuses on the application structure (logic, control, presentation, and meta-level management), application lifecycle, application adaptability and configurability, and provides functionality that allows altering the application structure at run-time. Furthermore, the Gaia application framework provides support for multi-modal interaction by simply attaching different types of Controllers to the applications (i.e., speech and gesture recognition).

10.3.6 Graspable and Tangible User Interfaces

Graspable Interfaces [87, 90] present an evolutionary model for GUIs where physical objects are used to interact with applications. This approach distinguishes time-multiplexed input devices from space-multiplexed input devices. The most well known example of a time-multiplexed input device is the mouse. The same device is multiplexed over time to control different GUI widgets. On the other hand, the space-multiplexing model is based on the idea of associating specific physical objects to specific functional aspects of the application. The objects become dedicated functional manipulators. There is a one-to-one mapping from a particular object to a virtual function. An example of space-multiplexed device is an audio mixing console, where each slider is associated with a specific music channel. The application framework I propose combines both concepts and defines the time-space-multiplexed model.

Tangible User Interfaces (TUI) [91] are a generalization of graspable interfaces. This new approach presents a model that combines digital and physical entities (bits and atoms). The main difference with graspable interfaces is that in the case of TUIs, physical objects can be both the input and output of the application. In general, TUIs propose moving the GUI from the desktop's monitor to the physical space inhabited by the user. Therefore, the world becomes the interface. TUI defines a new graphical user interface model called MCRpd (Model-Controller-Representation physical, digital). Tangible bits [92] are based on the traditional MVC. This model reuses the model and controller components from MVC and replaces the view component by the physical and digital representation. Audio and video are examples of digital representations, while physical entities such as "bricks" [93, 94]. This approach is solely based on using physical objects as controllers (physical objects are both representations and controllers). The Gaia application framework also considers the possibility of physical representations (externalizations) and physical controllers (introduced in the system by means of context input). However the framework allows using digital controllers which can be combined with the physical ones.

10.4 Computer-Supported Collaborative Work

Computer-Supported Collaborative Work (CSCW) refers to the research field that studies how to use computer technology to coordinate groups of people working on a common task. There are several CSCW systems and frameworks that provide functionality to allow groups of users to share media (e.g., video, audio, documents, and data) and interact, regardless of their physical location. In most cases, these systems provide tools to simplify the development of CSCW applications, by providing a middleware implementation that includes an application framework.

Different CSCW systems and frameworks target different aspects. I include next a list with some projects and their main properties.

MASH (Multimedia Architecture that Scales across Heterogeneous environments) [95] is a toolkit for multimedia communication over the Internet using the IP Multicast protocol. It is an outgrowth of the Internet MBone tools, and provides a number of applications for audio and video, as well as a framework to develop applications.

DISCIPLE [96] is a Java-Beans based framework that provides support for multimodal collaboration. Desktop users can interact via speech, gaze tracking, gestures, and traditional windows, and the system is able to switch between RPC and mobile code (agents) depending on the performance of the system.

Virtue/Orbit [97] are two software infrastructures that can be used independently or together for collaboration in data-intensive environments. Orbit provides support for supporting a number of users working on multiple collaborative tasks, while Virtue provides functionality to visualize and annotate massive data sets.

Wactlar et al. [98] present a system that uses speech, image, and natural language processing, combined with GPS to capture, integrate, and communicate personal multimedia experiences.

The Application Framework discussed in this thesis is not specifically customized to CSCW. Its main goal is to provide functionality to build active space aware applications, which not always imply collaborative work. The framework has built in support to register and remove applications components (via the coordinator) that belong to different users. Furthermore, applications can interact with the coordinator to obtain a list of users registered with the applications. The application framework provides the building blocks to construct specialized components (e.g., model and coordinator) to support the development of CSCW applications.

10.5 Application Mobility

There several platforms that provide support for application mobility among heterogeneous devices. I present in this section some of these platforms and compare them to the application framework. The main difference between the Gaia application framework and the rest of platforms is that the former considers the space as the device, and abstracts the devices it contains as execution nodes. The other platforms target individual devices.

iMASH (Interactive Mobile Application Support for Heterogeneous clients) [66] is a middleware infrastructure that allows moving applications among heterogeneous devices. The infrastructure implements filtering and caching mechanisms to accommodate different types of devices. iMASH distinguishes three different types of mobility[67]: one-way non-interactive (changes at the client are not propagated), one-way interactive (clients cannot modify original data but can add and modify additional data), and two-way interactive (clients can modify the original data and changes are propagated). Gaia and iMASH work at different abstraction levels. iMASH targets individual devices, while Gaia abstracts a space as a device. We could reuse the caching and transformation mechanisms presented by iMASH for inter-space mobility.

one.world [65] provides support for application mobility (migration) through a hybrid of process migration and application-dependent mobility. one.world addresses three aspects: migration has to be visible to applications, migration needs to integrate persistent storage, and migration has to be easy to control and centralized in the source code. The one.world architecture utilizes Virtual Machine based languages, such as Java and Microsoft Common Language Runtime. It organizes applications in environments, which contain all an application's data, both runtime state and persistent storage. Transfer of application state involves copying of the entire environment, which mirrors process migration. Runtime state is transferred via a checkpointed byte stream of the running application. Applications which have environmental-external dependencies are required to resolve and reconfigure when the environment is moved, thus paralleling application-dependent mobility. Gaia addresses the three aspects mentioned by one world. Migration visibility is achieved by notifying the model about migration, the *saveState* and *restoreState* methods concentrate all migration code, therefore simplifying the task to developers, and finally migration uses the Gaia context file system to have permanent access to data. The main difference between Gaia and one.world is that the latter assumes a single-device application model, as well as a homogeneous execution environment (i.e. virtual machines). Gaia mobility supports automatic application adaptation as applications move to heterogeneous environments.

Application mobility under Aura [64] uses a distributed file system to transfer information. Aura provides two simple abstractions for applications: Suppliers and Connectors. Suppliers provide the basic abstraction of data, such as text, while Connectors provide actual application interfaces to the data, such as Microsoft Word. By having Connectors on different platforms that work with the same Suppliers, Aura is able to offer transparent reconfiguration of the application. However, no explicit application management interface is offered to move applications to different (heterogeneous) active spaces.

Chapter 11

Conclusions

Ubiquitous computing environments such as active habitats and living spaces will become commonplace. The fast proliferation of wired and wireless networks, handheld devices and smart phones, sensors, and low-cost computers will make it possible. The key to enable these new environments relies on the ability to provide tools to simplify the development of applications that benefit from the active space abstraction. Furthermore, these applications must be easy to use and must hide the complexity of the underlying infrastructure to the end users. As a matter of fact, end users should not be concerned with device configuration issues and should concentrate on the functionality they want to obtain from the space.

In this thesis, we introduced the Active Space concept as an abstraction to address ubiquitous computing habitats and living spaces. We presented a meta-operating system to manage the Active Space as a programmable environment that hides resource heterogeneity, and introduced an application framework that relies on the meta-operating system and provides functionality to develop active space applications. These applications are resource-aware, space-independent, context-sensitive, user-centric, mobile, adaptive, and multi-device.

11.1 Contributions

The main contributions of this thesis are the abstraction of a physical space and the digital resources it contains as a programmable environment called an Active Space, a meta-operating system kernel to enable active spaces, and an application framework that supports the development of applications customized for this environment. The active space extends the computing environment associated with a computer to the space level and includes context as a fundamental computational parameter. Active spaces simplify the management of the resources and the development of applications.

As a result of deploying the meta-operating system in a real environment, we have contributed with fourteen active space-aware applications that assist users in a number of tasks, including audio playing, slide show presentations, and activity scheduling.

11.2 Perspective

We have been experimenting with a prototype active space for the last two years. During this time and thanks to the active space abstraction, the prototype active space has evolved from an "electronic warehouse" to an integrated computing environment that effectively coordinates all resources and hosts the execution of useful applications. However, although current results are promising, we are still beginning to understand the full potential of ubiquitous computing and active spaces. As pointed out by Christopher Lueg [99], there is an important difference between what is feasible and what really makes sense. The active space approach does not propose a reactive environment that understands human behavior and automatically reacts to assist the user. Instead, it provides the mechanisms to transform spaces into programmable entities. We believe that the contributions presented in this thesis are an interesting approach to tackle the complexity of ubiquitous computing environments.

11.3 Known Issues

The application framework original design tried to follow the MVC design as close as possible. However, after working on active spaces for a couple of years, we became more familiar with the overall environment, its problems, and its benefits. As a result, we modified the original design to accommodate the new application requirements. Section 10.3.1, presents a description of the main differences between MVC and the proposed framework.

There are at least two aspects we would like to change in the application framework. The first one is using adapters to adapt the interface of the model to different presentations. The goal is similar to the adapter used with the controller component. It would allow decoupling presentations from models. The adapter would include translations from the model events into messages the presentation understands, and would also map requests from the presentation into methods implemented by the model. Current implementation assumes that the presentations understand the messages sent by the model, and they know the interface of the model. As the number of applications grows, this may not be the case and might force developers to create customized presentations for each model.

The second aspect I would change in the framework is the notification mechanism implemented by the model. Currently, the model uses the Gaia OS Event Manager to create the channels to notify the presentations and controllers. However, due to the Event Manager implementation, if the kernel crashes, all channels are lost, and therefore all applications loose the ability to notify its presentations and controllers. Furthermore, if a channel crashes, it is not possible to re-start it and reattach previous listeners and suppliers. This issue has become a major concern in our prototype active space. When a channel crashes or we have to re-start the kernel, we are forced to terminate all applications and re-start them. This scenario is obviously unrealistic for an active space environment where errors are a known fact (e.g., network disconnections, devices removed from the active space, and crashing components). I propose extending the implementation of the model, so it implements the notification mechanism. In this way, the model becomes responsible for maintaining soft-state about presentations and controllers that require notifications, can implement different recovery policies, and becomes independent of problems in the kernel. Furthermore, implementing the notification mechanism in the model would allow developers to use different techniques such as IP multicast, broadcast, or RPC events.

Chapter 12

Future Work

This thesis contributed to understand the issues related to the development of applications for ubiquitous computing environments. The current application framework is the result of a number of design, implementation, and evaluation iteration cycles. This iterative process has been essential to evolve the framework and make it generic enough to support different types of applications. However, based on the same iterative principle, there is still room for improvement and additional functionality that will assist developers even further in the development of applications. I present in this chapter, a number of topics for future research.

12.1 Domain Specific Framework Extensions

We have used the application framework to build ten applications for our prototype active space. Although the framework was generic enough to support the development of all applications, we were able to classify the applications into two groups based on the nature of the notifications sent by the model to the presentations. The two groups are multicast and broadcast applications.

In a broadcast application, changes in the model affect all the presentations. For example, in the PDF Viewer application, when the user selects "next page" in the controller, the model sends a notification that affects all presentations – all presentations display the next page of the document.

In a multicast application, changes in the model affect only some presentations. For example, the Presentation Manager application provides functionality to display a number of synchronized slides in different displays. When the user selects “next” in the presentation, the model uses the synchronization rules defined by the user to send a text notification that includes a list of presentation ids and the slide number each presentation has to display. Presentations parse the notification and if their id is included they get the slide number and render it.

Broadcast applications do not require any additional functionality to send notifications to the presentations. The default framework’s model implements a broadcast notification mechanism. However, multicast applications require a specialized model that encodes the id of the target presentations in the notification message. Based on this observed behavior, it is possible to provide a customized model that implements the multicast notification mechanism.

As part of the future research we plan to implement a collection of models and presentations customized to different types of applications, including notification multicast, streaming, and workgroup. This extensions subclass the default framework components and simplify application development by providing default mechanisms.

12.2 Fault Tolerance

Current application framework implementation provides support for reliability based on user configurable policies. The default implementation detects a faulty component and automatically removes it from the application. Furthermore, current implementation only monitors presentations and controllers. We plan to investigate and develop new fault tolerance policies that automatically restart crashing components. New policies will take into account the application model and will periodically save its state so it is possible to restart the model and inject the previously saved state. Finally, in order to take into account coordinator fault tolerance, we plan to develop a service that monitors the status of all coordinators running in an active space. The service periodically saves the state of the coordinator so it can restart it and bring it back to the previously saved state.

12.3 Application Mobility

During our experiments with inter-active space application mobility[62] we implemented a number of different mobility policies. The diversity of applications, active spaces, and application utilization patterns indicates that a single mobility mechanism in not enough. In some cases a user may want to move the whole application from one space to another, in some other cases the user may decide to move only the presentations to another space (e.g., shared application).

We plan to research intra-active space mobility policies. These policies will provide common intra-active space mobility patterns based on context, application and active space type, and user preferences. For example, a mobility policy may move the VCR controller of the presentation manager application automatically based on the user location. These policies will help customizing the behavior of the active space applications to the user preferences.

12.4 Using Application Bridges to Alter the Meta-Level of the Applications

We plan to work on more sophisticated bridges that leverage the low-level functionality provided by the Gaia OS (e.g., context, presence, and security) and interact with the application coordinator to alter the meta-level. All current bridges alter the base-level functionality of the target application (application functionality domain). We plan to extend our experiments with bridges that affect the meta-level of the target application (interacting with the coordinator of the target application). For example, a bridge between the location and the music application can move the audio from the room speakers to the user's laptop when it detects that the user is not alone, and can move the audio back to the room when everybody else leaves.

12.5 Extending the Application Framework to Support Multimedia Applications

As described in section 3.1, the application framework presented in this thesis does not support the development of multimedia applications. The architecture of these applications demands a distributed implementation of the model and new services and protocols to deal with issues such as deployment, monitoring, path creation, mapping, and adaptation of the services composing the model. These applications are normally described in terms of a Task Flow model (directed acyclic graph) that represents the dependencies among the different services that implement the application. Furthermore, new mechanisms are required to map the task flow to physical resources, monitor the QoS parameters, and adapt the application whenever changes are introduced in the application. The main difference between a multimedia and a non-multimedia application is in the model of the application. In a multimedia application, the model is implemented as a collection of services, while a non-multimedia application, the model is most of the times a single service. Figure 45 illustrates the difference.

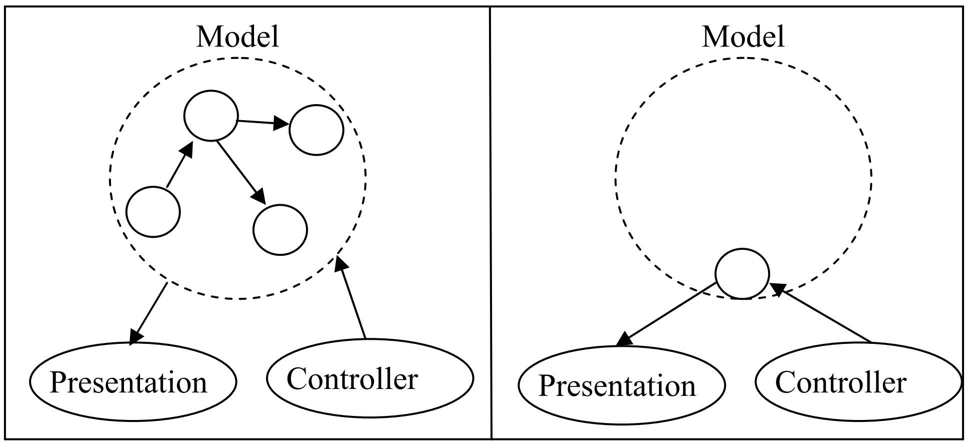


Figure 45. Multimedia Application (left) vs. Non-multimedia Application (right).

As part of the future work we propose extending the application framework to provide support for multimedia applications. This modification would allows us to have a single framework for both multimedia and non-multimedia applications, which would allows us to have common protocols for both types of applications, and most importantly, a unified application development framework. Figure 46 illustrates the proposed extensions to the original framework to deal with multimedia applications. There are two major changes: a specialized coordinator (Composer) and a new component we refer to as Processing Unit. The Composer is a subclass of the generic Coordinator that extends it with additional QoS functionality, as well as information about the task flow model that underlies the multimedia application. The Processing Unit (PU) is a component that implements some model related functionality. PUs are generic base classes for multimedia components such as encoders, decoders, transcoders, and streamers, and are composed, monitored, and rearranged according to the application's task flow enforced by the composer.

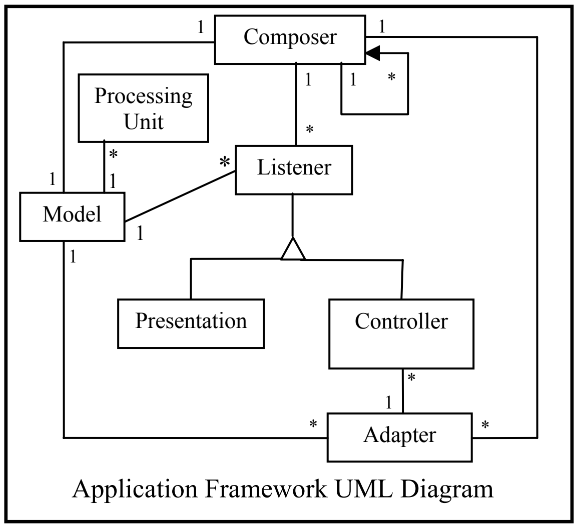


Figure 46. Proposed QoS Aware Application Model.

References

[1] Mark Weiser, "The Computer for the Twenty-First Century," in *Scientific American*, vol. 265, 1991, pp. 94-101.

[2] Hewett, Baecker, Card, Carey, Gassen, Mantei, Perlman, Strong, and Verplank, "ACM SIGCHI Curricula for Human-Computer Interaction," ACM SIGCHI (http://sigchi.org/cdg/cdg2.html), 2002.

[3] Michi Henning and Steve Vinosky, *Advanced CORBA Programming with C++*: Addison-Wesley, 1999.

[4] Henry Bequet, *Professional Java SOAP*: Wrox Press, 2001.

[5] Rickard Oberg, *Mastering RMI*: Wiley, 2001.

[6] Bhaskarjyoti Borthakur, "Distributed and Persistent Event System For Active Spaces," in *Master Thesis in Computer Science*. Urbana-Champaign: University of Illinois at Urbana-Champaign, 2002, pp. 67.

[7] Bill N. Schilit, Norman Adams, and Roy Want, "Context-Aware Computing Applications," Proceedings of IEEE Workshop on Mobile Computing Systems and Applications, 1994.

[8] Anind K. Dey, Daniel Salber, and Gregory D. Abowd, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," *Human-Computer Interaction (HCI)*, vol. 16(2-4), pp. 97-166, 2001.

[9] Jason I. Hong and James A. Landay, "An Infrastructure Approach to Context-Aware Computing," *Human Computer Interaction*, vol. 16(4), 2001.

[10] Steven A. N. Shafer, Barry Brumitt, and JJ Cadiz, "An Infrastructure Approach to Context-Aware Computing," *Human Computer Interaction*, vol. 2001(16), 2001.

[11] Paul Castro and Richard Muntz, "Managing Context for Smart Spaces," in *IEEE Personal Communications*, vol. 7, 2000, pp. 44-46.

[12] Anind K. Dey and Gregory D. Abowd, "The Context Toolkit: Aiding the Development of Context-Aware Applications," Proceedings of Workshop on Software Engineering for Wearable and Pervasive Computing,Limerick, Ireland, 2000.

[13] Robert McGrath and Dennis Mickunas, "Discovery and Its Discontents: Discovery Protocols for Ubiquitous Computing," University of Illinois at Urbana-Champaign, Urbana, Technical Report UIUCDCS-R-99-2132, March 2000.

[14] M. Wahl, T. Howes, and S. Kille, "Lightweight Directory Access Protocol (v3)," vol. 1997: IETF RFC 2251, 1997.

[15] Salutation Consortium, "Salutation," http://www.salutation.org.

[16] Microsoft, "Understanding Universal Plug and Play (http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc)."

[17] E. Guttman, C. Perkins, J. Veizades, and M. Day, "Service Location Protocol, version 2," vol. 1999: IETF, RFC 2608, http://www.rfc-editor.org/rfc/rfc2608, 1999.

[18] W. Keith Edwards, *Core Jini*: The Sun Microsystems Press, 1999.

[19] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz, "An Architecture for a Secure Service Discovery Service," Proceedings of Mobile Computing and Networking, pp.24-35, 1999.

[20] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr., "Semantic File Systems," Proceedings of SOSP13, pp.16-25, 1991.

[21] Christopher K. Hess, "A Context File System for Ubiquitous Computing Environments," University of Illinois at Urbana-Champaign, Urbana-Champaign, CS Technical Report UIUCDCS-R-2002-2285 UILU-ENG-2002-1729, July 2002 2002.

[22] Brad A. Myers, "Using Hand-Held Devices and PCs Together," in *Communications of the ACM*, vol. 44, 2001, pp. 34-41.

[23] Glenn E. Krasner and Stephen T. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System," *Journal of Object Oriented Programming*, vol. 1(3), pp. 26-49, 1988.

[24] Renato Cerqueira, Carlos Cassino, and Roberto Ierusalimschy, "Dynamic component gluing across different componentware systems," Proceedings of International Symposium on Distributed Objects and Applications (DOA'99), pp.362-371, Edinburgh, 1999.

[25] Roberto Ierusalimschy, Luiz Figuereido, and Waldemar Celes, "Lua: An Extensible extension language," Proceedings of Software: Practice and Experience, pp.635-652, 1996.

[26] Avi Silbershatz and Peter Galvin, *Operating System Concepts*, 5 ed: Addison Wesley, 1998.

[27] Fabio Kon, Ashish Singhai, Roy H. Campbell, Dulcineia Carvalho, Robert Moore, and Francisco J. Ballesteros, "2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments," Proceedings of ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, pp.388-389, Brussels, Belgium, 1998.

[28] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros, "2K: A Distributed Operating System for Dynamic Heterogeneous Environments," Proceedings of 9th IEEE International Symposium on High Performance Distributed Computing,Pittsburgh, 2000.

[29] Ashish Singhai, Aamod Sane, and Roy H. Campbell, "Quarterware for Middleware," Proceedings of 18th IEEE International Conference on Distributed Computing Systems (ICDCS 1998), pp.192-201, Amsterdam, The Netherlands, 1998.

[30] Geoff Coulson, Gordon Blair, Nigel Davies, Philippe Robin, and Tom Fitzpatrick, "Supporting Mobile Multimedia Applications through Adaptive Middleware," *IEEE Journal on selected areas in Communications*, vol. 17(9), pp. 1651-1659, 1999.

[31] Fabio Kon, Fabio Costa, Roy Campbell, and Gordon Blair, "The Case for Reflective Middleware," *Communications of the ACM*, vol. 45(6), pp. 33-38, 2002.

[32] Fabio Kon, Manuel Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhaes, and Roy H. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000),New York, 2000.

[33] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, "The Design of the TAO Real-Time Object Request Broker," *Computer Communications. Elsevier Science*, vol. 21(4), 1998.

[34] Manuel Roman, Ashish Singhai, Dulcineia Carvalho, Christopher Hess, and Roy H. Campbell, "Integrating PDAs into Distributed Systems: 2K and PalmORB," Proceedings of International Symposium on Handheld and Ubiquitous Computing (HUC'99),Karlsruhe, Germany, 1999.

[35] Manuel Roman, Dennis Mickunas, Fabio Kon, and Roy H. Campbell, "LegORB and Ubiquitous CORBA," Proceedings of IFIP/ACM Middleware'2000 Workshop on Reflective Middleware,IBM Palisades Executive Conference Center, NY, 2000.

[36] Manuel Roman, Fabio Kon, and Roy H. Campbell, "Reflective Middleware: From Your Desktop to Your Hand," *IEEE Distributed Systems Online. Special Issue on Reflective Middleware*, 2001.

[37] G. D. Abowd, "Classroom 2000: An experiment with the instrumentation of a living educational environment," *IBM Systems Journal*, vol. 38(4), pp. 508-530, 1999.

[38] Anind K. Dey, "CyberDesk: A Framework for Providing Self-Integrated Context-Aware Services," *Knowledge-Based Systems*, vol. 11(1), pp. 3-13, 1998.

[39] Anind K. Dey, Gregory D. Abowd, and Daniel Salber, "A Context-Based Infrastructure for Smart Environments," Proceedings of Workshop on Managing Interactions in Smart Environments (MANSE), 1999.

[40] Michael L. Dertouzos, "The Future of Computing," in *Scientific American*, 1999.

[41] Michael Coen, Brenton Phillips, Nimrod Warshawsky, Luke Weisman, Stephen Peters, and Peter Finin, "Meeting the Computational Needs of Intelligent Environments: The Metaglue System," Proceedings of MANSE'99,Dublin, Ireland, 1999.

[42] Armando Fox, "Building Scalable, Composable, Adaptive Internet Services with TACC," in *PhD Thesis in EECS*. Berkeley: University of Berkeley, 1998.

[43] Manuel Roman and Roy H. Campbell, "GAIA: Enabling Active Spaces," Proceedings of 9th SIGOPS European Workshop,Kolding, Denmark, 2000.

[44] Shankar R. Ponnekanti, Brad Johanson, Emre Kiciman, and Armando Fox, "Portability, Extensibility, and Robustness in iROS," Proceedings of PerCom 2003,Dallas-Fort Worth, Texas, USA, 2003.

[45] Robert Grimm, Janet Davis, Eric Lemar, Adam McBeath, Steven Swanson, Steven Gribble, Tom Anderson, Brian Bershad, Gaetano Borriello, and David Wetherall, "Programming for Pervasive Computing Environments," University of Washington, Technical Report: UW-CSE-01-06-01, Washington 2001.

[46] Duangdao Wichadakul, Xiaohui Gu, and Klara Nahrstedt, "A Programming Framework for Quality-Aware Ubiquitous Multimedia Applications," Proceedings of ACM Multimedia 2002,Juan Les Pins, France, 2002.

[47] Manuel Roman and Roy H. Campbell, "Providing Middleware Support for Active Space Applications," Proceedings of ACM/IFIP/USENIX International Middleware Conference,Rio de Janeiro, Brazil, 2003.

[48] Simon Brown, Robert Burdick, Jayson Falkner, Ben Galbraith, Rod Johnson, Larry Kim, Casey KOchmer, Thor Kristmundsson, and Sing Li, *Professional JSP*, 2nd ed: Wrox Press Inc., 2001.

[49] Andrew Harbourne-Thomas, Sam Dalton, Simon Brown, Bjarki Holm, Tony Loton, Meeraj Kunnumpurath, Subrahmanyam Allamaraju, John Bell, and Sing Li, *Professional Java Servlets 2.3*, 1st ed: Wrox Press Inc., 2002.

[50] Sue Spielman, *The Struts Framework: Practical Guide for Programmers*, 1st ed: Morgan Kaufmann Publishers, 2002.

[51] Klara Nahrstedt, Dongyan Xu, Duangdao Wichadakul, and Baochun Li, "QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments," *IEEE Communications*, vol. 39(11), 2001.

[52] Xiaohui Gu, Klara Nahrstedt, Alan Messer, Ira Greenberg, and Dejan Milojicic, "Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments," Proceedings of IEEE Percom 2003,Dallas-Fort Worth, Texas, 2003.

[53] Xiaohui Gu, Klara Nahrstedt, Wanghong Yuan, Duangdao Wichadakul, and Dongyan Xu, "An XML-based Quality of Service Enabling Language for the Web," *Journal of Visual Language and Computing (JVLC), special issue on Multimedia Languages for the Web*, vol. 13(1), pp. 61-95, 2002.

[54] Tim O'Reilly, Mark Langley, and David Flanagan, *X Toolkit Intrinsic Reference Manual for Version 11 of the Window System*: O'Reilly, 1995.

[55] Gregory D. Abowd and Elizabeth D. Mynatt, "Charting Past, Present, and Future Research in Ubiquitous Computing," *ACM Transactions on Computer-Human Interaction*, vol. 7(1), pp. 29-58, 2000.

[56] Gregor Kiczales, Jim des Rivires, and Daniel G. Bobrow, "The Art of the Metaobject Protocol," *MIT Press*, 1991.

[57] Manuel Roman, Fabio Kon, and Roy H. Campbell, "Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO case," Proceedings of ICDCS,Austin, Texas, 1999.

[58] Fabio M. Costa, Gordon S. Blair, and Geoff Coulson, "Experiments with an architecture for reflective middleware," *IOS Press*, vol. 7(3), pp. 313-325, 2000.

[59] Anind K. Dey, "Providing Architectural Support for Building Context-Aware Applications," in *PhD Thesis in Computer Science*. Atlanta: Georgia Institute of Technology, 2000, pp. 188.

[60] Daniel Salber, Anind K. Dey, and Gregory D. Abowd, "The Context Toolkit: Aiding the Development of Context-Enabled Applications," Proceedings of CHI'99, pp.15-20, Pittsburgh, 1999.

[61] Manuel Roman, Brian Ziebart, and Roy H. Campbell, "Dynamic Application Composition: Customizing the Behavior of an Active Space," Proceedings of PerCom2003,Dallas-Fort Worth, Texas, USA, 2003.

[62] Herbert Ho, "Application Mobility in Active Spaces," in *Master Thesis in CS*. Urbana: University of Illinois at Urbana-Champaign, 2002, pp. 52.

[63] Manuel Roman, Herbert Ho, and Roy H. Campbell, "Application Mobility in Active Spaces," Proceedings of 1st International Conference on Mobile and Ubiquitous Multimedia, pp.66-76, Oulu, Finland, 2002.

[64] Joao Pedro Sousa and David Garlan, "Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments," Proceedings of IEEE/IFIP Conference on Software Architecture, pp.29-43, Montreal, 2002.

[65] Robert Grimm, Janet Davis, Eric Lemar, and Brian Bershad, "Migration for Pervasive Applications," 2002.

[66] Thomas Phan, Richard Guy, Jing Gu, and Rajive Bagrodia, "A Scalable, Distributed Middleware Service Architecture to Support Mobile Internet Applications," Proceedings of Workshop on Wireless Mobile Internet (WMI 2001),Rome, Italy, 2001.

[67] Thomas Phan, Richard Guy, Jing Gu, and Rajive Bagrodia, "A New TWIST on Mobile Computing: Two-Way Interactive Session Transfer," Proceedings of Workshop on Internet Applications (WIAPP 2001), pp.2-12, San Jose, CA, 2001.

[68] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell, "The Case for Reflective Middleware," *Communications of the ACM*, vol. 45(6), pp. 33-38, 2002.

[69] Microsoft, "Terminal Services," (http://www.microsoft.com/windows2000/technologies/terminal/default.asp), 2002.

[70] Gregor Kiczales, "Beyond the Black Box: Open Implementation," *IEEE Software*, vol. 13(1), pp. 137-142, 1996.

[71] Brian Cantwel Smith, "Reflection and Semantics in LISP," Proceedings of 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp.23-25, Salt Lake City, Utah, United States, 1984.

[72] Nullsoft, "Winamp Player (http://www.winamp.com)."

[73] Dale Rogerson, *Inside COM*: Microsoft Press, 1997.

[74] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven Shafer, "EasyLiving: Technologies for Intelligent Environments," Proceedings of Handheld and Ubiquitous Computing (HUC), pp.12, Bristol, England, 2000.

[75] Peter Tandler, "Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices," Proceedings of Ubicomp 2001: Ubiquitous Computing, pp.96-115, Atlanta, Georgia, 2001.

[76] Norbert Streitz, Jorg Geissler, and Torsten Holmer, "Roomware for Cooperative Buildings: Integrated Design of Architectural Spaces and Information Spaces," Proceedings of Workshop on Cooperative Buildings (CoBuild'98), pp.4-21, Darmstad, Germany, 1998.

[77] Armando Fox, Brad Johanson, Pat Hanrahan, and Terry Winograd, "Integrating Information Appliances into an Interactive Workspace," *IEEE Computer Graphics & Applications*, vol. 20(3), 2000.

[78] Brad Johanson, Armando Fox, and Terry Winograd, "Experiences with Ubiquitous Computing Rooms," *IEEE Pervasive Computing Magazine*, vol. 1(2), pp. 67-74, 2002.

[79] Shankar R. Ponekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd, "ICrafter: A Service Framework for Ubiquitous Computing Environments," Proceedings of Ubicomp 2001: Ubiquitous Computing, pp.56-75, Atlanta, Georgia, 2001.

[80] Scott M. Thayer and Peter Steenkiste, "An Architecture for the Integration of Physical and Informational Spaces," Proceedings of ARCS'02, pp.7-20, Karlsruhe, Germany, 2002.

[81] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste, "Project Aura: Towards Distraction-Free Pervasive Computing," in *IEEE Pervasive Computing*, vol. 1, 2002, pp. 22-31.

[82] Manuel Roman, James Beck, and Alain Gefflaut, "A Device-Independent Representation for Services," Proceedings of Workshop on Mobile Computing Systems and Applications,Monterey, CA, USA, 2000.

[83] J. Eisenstein, J. Vanderdonckt, and A. Puerta, "Adapting to Mobile Contexts with User-Interface Modeling," Proceedings of Workshop on Mobile Computing Systems and Applications,Monterey, CA, USA, 2000.

[84] Guruduth Banavar, James Beck, Eugene Gluzberg, Jonathan Munson, Jeremy B. Sussman, and Debory Zukowski, "An Application Model for Pervasive Computing," Proceedings of 6th ACM MOBICOM, pp.266-274, Boston, MA, 2000.

[85] Joelle Coutaz, "PAC, an Object Oriented Model for Dialog Design," Proceedings of Human Computer Interaction. INTERACT, pp.431-436, 1987.

[86] Ralph D. Hill, "The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interface to Applications," Proceedings of CHI, 1992.

[87] George W. Fitzmaurice, "Graspable User Interfaces," in *PhD Thesis in Computer Science*. Toronto: University of Toronto, 1996.

[88] Duangdao Wichadakul, Klara Nahrstedt, Xiaohui Gu, and Dongyan Xu, "2KQ+: An Integrated Approach of QoS compilations and Component-Based, Run-Time Middleware for the Unified Qos Management Framework," Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001),Heidelberg, Germany, 2001.

[89] Brad Myers, Robert Malkin, Michael Bett, Alex Waibel, Ben Bostwick, Robert C. Miller, KJie Yang, Matthias Denecke, Edgar Seeman, Jie Zhu, Choon Hong Peck, Dave Kong, Jeffrey Nichols, and Bill Scherlis, "Flexi-Modal and Multi-Machine User Interfaces," Proceedings of Fourth International Conference on Multimodal Interfaces,Pittsburgh, PA, 2002.

[90] R. Poynor, "The hand that rocks the cradle," in *The International Design Magazine*, vol. May-June, 1995.

[91] Hiroshi Ishii and Brygg Ullmer, "Tangible bits: Towards seamless interfaces between people, bits and atoms," Proceedings of Conference on Human Factors in Computing Systems, pp.234-241, Atlanta, 1997.

[92] Brygg Ullmer and Hiroshi Ishii, "Emerging frameworks for tangible user interfaces.," *IBM Systems Journal*, vol. 39(3&4), 2000.

[93] B. Ullmer, H. Ishii, and D. Glas, "MediaBlocks: Physical container, transports, and controls for online media," Proceedings of SIGGRAPH'98, pp.379-386, New York, 1998.

[94] George W. Fitzmaurice, Hiroshi Ishii, and William Buxton, "Bricks: Laying the Foundations for Graspable User Interfaces," Proceedings of CHI, 1995.

[95] Open Mash Consortium, "Open Mash," http://www.openmash.org/, 2002.

[96] Ivan Marsic, "DISCIPLE: A Framework for Multimodal Collaboration in Heterogeneous Environments," *ACM Computing Surveys*, vol. 31(2), 1999.

[97] Daniel A. Reed and Simon M. Kaplan, "Orbit/Virtue:Collaboration and Visualization Toolkits," *ACM Computing Surveys*, vol. 31(2), 1999.

[98] Howard D. Wactlar, Michael G. Christel, Alexander G. Hauptmann, and Yigong Gong, "Informedia Experience-on-Demand: Capturing, Integrating and Communicating Experiences Across People, Time and Space," *ACM Computing Surveys*, vol. 31(9), 1999.

[99] Christopher Lueg, "On the Gap Between Vision and Feasibility," Proceedings of Pervasive 2002, pp.45-57, Switzerland, 2002.

Vita

Manuel Roman was born in Barcelona, Spain, in 1974. He received a “Ingeniero Informatico” degree in 1997 from the Ramon Llull University (La Salle School of Engineering). His research interests include Ubiquitous Computing, Distributed Systems, Operating Systems, Middleware, and frameworks for handheld devices. He has published over 20 papers on these topics.

Manuel was one of the main architects of the Gaia OS, a meta-operating system for active spaces. Furthermore, Manuel has contributed in the development of dynamicTAO, a dynamically reconfigurable ORB, and has developed the Universally Interoperable Core, a fully reconfigurable middleware platform designed for embedded devices with limited resources.