A Middleware-Based Application Framework for Active Space Applications

Manuel Román and Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{mroman1, rhc}@cs.uiuc.edu

Abstract. Ubiquitous computing challenges the conventional notion of a user logged into a personal computing device, whether it is a desktop, a laptop, or a digital assistant. When the physical environment of a user contains hundreds of networked computer devices each of which may be used to support one or more user applications, the notion of personal computing becomes inadequate. Further, when a group of users share such a physical environment, new forms of sharing, cooperation and collaboration are possible and mobile users may constantly change the computers with which they interact; we refer to these digitally augmented physical spaces as Active Spaces. We present in this paper an application framework that provides mechanisms to construct, run or adapt existing applications to ubiquitous computing environments. The framework binds applications to users, uses multiple devices simultaneously, and exploits resource management within the users' environment that reacts to context and mobility. Our research contributes to application mobility, partitioning and adaptation within device rich environments, and uses context-awareness to focus the resources of ubiquitous computing environments on the needs of users.

1. Introduction

Future ubiquitous computing will surround users with a comfortable and convenient information environment that merges physical and computational infrastructures into an integrated habitat. Context-awareness should accommodate the habitat to the user preferences and tasks, group activities, and the nature of the physical space. We term this dynamic and computational rich habitat an Active Space. Within the space, users will interact with flexible applications that may move with the user, may define the function of the habitat, or collaborate with remote applications. The research described in this paper builds on experiments with applications conducted in a prototype active meeting room (Figure 1). We have currently developed fourteen applications that we use regularly in our seminars, meetings, and presentations.

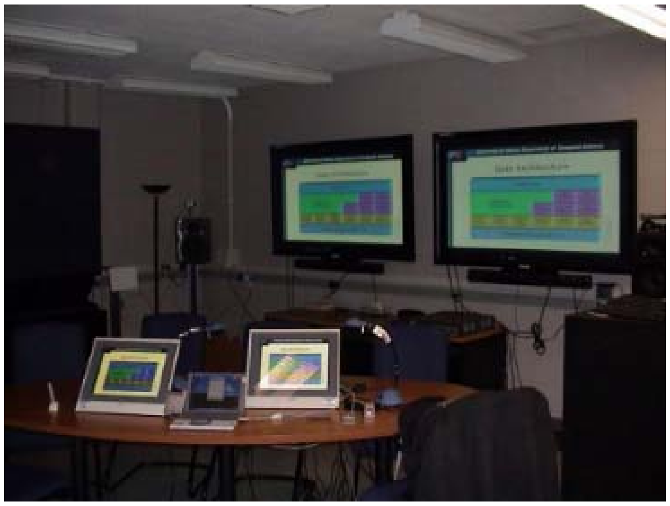

Fig. 1. Prototype Active Meeting Room Hosting a Slide Show Application

The Active Space consists of the Gaia middleware OS[1] managing a distributed system composed of four 61" wall-mounted plasma displays, a video wall, 5.1 audio system (Dolby Digital), touch screens, IR beacons, badge detectors, and wireless and wired networks connecting 15 Pentium-4 PCs running Windows 2000 and Windows CE based Compaq iPaq PDAs. Gaia supplies services including event delivery, entity presence detection (devices, users, and services), context notification, a space repository to store information about entities present in the space, and a context-aware file system.

The application experiments examine how to construct applications that use multiple devices simultaneously, take advantage of resources contained in the user habitat, exploit context information (e.g., location and social activity), benefit from automatic data transformation and can alter their composition dynamically (e.g., attaching and detaching components) to adapt to changes in the Active Space, and move with the users to different Active Spaces.

The problem we focus in this paper consists on providing an application framework that leverages the functionality provided by the Gaia middleware OS to assist developers in the construction of Active Space application. The application framework addresses three issues: (1) defining an application model 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 flexible policy driven application management interface that allows customizing applications to the dynamic behavior of Active Spaces.

The paper continues with a description of the issues we consider are key for Active Space applications (Section 2), a description of the application framework including information about the application model (Section 3), the mapping process (section 4), and the application management functionality (Section 5). Section 6 explains how the application framework addresses the issues listed in Section 2, Section 7 presents an example of an application we have built using the framework, and Section 8 discusses performance evaluation. We present related work in Section 9, and conclude in Section 10.

2. Active Space Applications' Key Issues

Based on our experiments, we define an Active Space application as a collection of dynamically assembled components that fulfill the requirements of a user or a group of users. Dynamism is probably the most important aspect of an Active Space application, and requires a flexible component based application architecture capable of changing its own composition at run-time. We have identified a number of issues that are common to most Active Space applications. These issues are the cornerstones of our application framework, which effectively simplifies the development of Active Space applications. We list these issues next.

2.1. Resource-Awareness

Ubiquitous computing scenarios contain hundreds of resources, including devices (e.g., sensors, displays, and CPUs), services (e.g., file management, printing, and temperature controller), and applications (e.g., slideshow presenter, music player, and calendar). In order to exploit these resources, Active Spaces must provide functionality to discover existing resources, functionality to store information about resources including their capabilities, their availability, and their cardinality, and functionality to query for specific resources.

2.2. Multi-Device

In an environment where users are surrounded by hundreds of devices, the notion of interacting with a single device becomes inappropriate. Users may utilize different devices at different times, or may use multiple devices simultaneously to accomplish a well defined goal, 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) 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 proposed by Myers et al. [2], 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.

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 users' current environment.

Abowd et. al.[3] 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.

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 [4-8].

Adaptation of the interactive components' composition (altering the number and location of the components the user utilizes to interact with the application) does not apply to traditional interactive applications running on desktops due to, at least, three main reasons:

1. 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.
2. 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.
3. 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 another user is entering, the application moves the document to the user's personal PDA to protect confidentiality. This requires attaching a new application component (the one on the PDA) and removing an existing one (the one in the display).

2.5. Mobility

Application partitioning and user-centrism require applications to be mobile. There are at least 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. Inter-space mobility concerns moving applications across different spaces, and is a consequence of user-centrism (users are mobile by definition).

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 to the space as a default computational parameter. Context is one of the most important properties in ubiquitous computing [9] and therefore applications must 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.

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 for each environment. Users must be able to use the same applications in their active home, active car, and active office.

3. Application Model

We have implemented an application framework that simplifies the development of applications for Active Spaces. The application framework models applications as a collection of distributed components, reuses the application partitioning proposed by the Model-View-Controller pattern[10], and covers all the aspects presented in Section 2. The application framework is implemented on top of a Middleware Operating System (Gaia OS), defines an application model, implements functionality for application mapping, and implements a number of application management protocols. In this section, we present the application model and describe the application mapping, and the management protocols in the following sections.

The application model consists of five components: 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 2 illustrates the application model.

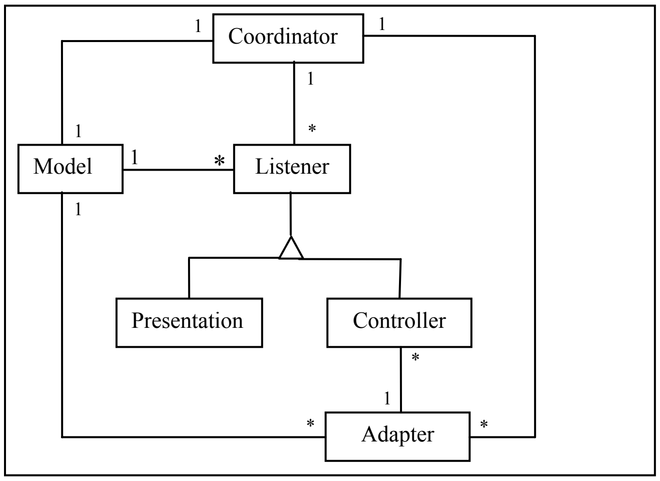

Fig. 2. Application Model UML Diagram

3.1. Model

The Model component implements the logic of the application, stores and synchronizes the application's state, and provides an interface to access the application functionality. The Model maintains a list of listeners and it is responsible for notifying them about changes in the application's state to keep them synchronized. There is no restriction on the implementation of the Model, which can be built as a single component or as a collection of distributed components. 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

3.2. 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 the MVC, which was originally defined as a graphical representation rendered on a display. An important difference with MVC views is that presentations are output entities and do not handle user inputs. This behavior is required to model non-graphical presentations such as a music player, which cannot coordinate input events. Presentations are implemented as listeners that can be attached to and detached from the Model dynamically. 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 (attach) 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 middleware infrastructure invokes the *detach* method on the Presentation so the Presentation stops presenting the application's data and releases 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 implementation of the notify method is Presentation dependent; however, the common behavior consists on retrieving the new application state from the Model (using the Model's interface) and updating the Presentation's data, which affects the output perceived by the users.

3.3. Controller

A Controller is a component (i.e., hardware and software) capable of altering the application's state through the Model's interface. Examples of hardware controllers are mice, keyboards, and active badges. Examples of software controllers are GUIs (e.g., MVC and PAC[11] based) 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. Encapsulating context in controllers has all the benefits described by Salber et. al. [12], and simplifies the development of applications that can easily react to changes in the context.

Controllers are implemented as Model listeners and therefore receive notifications from the Model (notify method) 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 ignore the notifications. Similarly to presentations, controllers implement *attach*, *detach*, and *notify* which are invoked when the Controller is attached to, detached from, and notified by the Model.

3.4. Adapter

This component coordinates the interaction between controllers and the application Model. It maps method calls generated by controllers into requests to the application Model dynamically, therefore decoupling controllers from specific models.

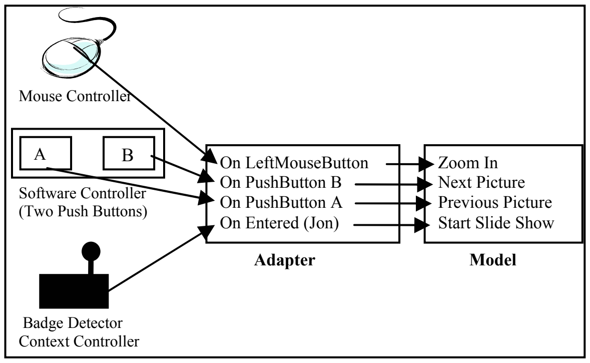

Fig. 3. Adapter Example.

Figure 3 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 using the *setMapping* method.

According to the application model, it is possible to associate more than one Adapter with the same application. Depending on configurable properties (e.g., type of Controller, user utilizing the Controller, or context properties such as location) different adapters can be activated at different times, therefore changing the effect of controllers on the application.

3.5. Coordinator

Active space applications are a collection of distributed components composed of a Model and a number of presentations, controllers, and adapters. 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.

The Coordinator encapsulates information about the application components' composition (i.e., application meta-level) and provides an interface to register and unregister presentations, controllers, and adapters. The Coordinator provides also functionality to retrieve run-time information about the composition of the application components, 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. 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, and can be overridden by users.

4. Application Mapping

The proposed application mapping mechanism provides functionality to build applications that can be used in heterogeneous Active Spaces.

Applications based on the application framework are independent of a particular Active Space by using generic application descriptions that list the application components and their requirements. These descriptions are used to create a specific application description that uses resources present in the Active Space, which match the application requirements listed in the generic description. The application framework defines two types of application descriptions: the application generic description (AGD), and the application customized description (ACD).

The AGD (Figure 4, left) is an Active Space-independent application description that lists the components of an application and their requirements. The AGD uses name-value pairs to describe the component's requirements and it is used as a template from which concrete application configurations (i.e., ACDs) are generated. The description contains a list of application components consisting of one Model, one Coordinator, zero or more presentations, and zero or more controllers. Every component entry includes a component name, an optional field with the parameters required, a field with the component cardinality (minimum and maximum number of instances of the component allowed), and a list of requirements for the component, which include information such as for example, required operating system, and hardware platform. The mapping mechanism uses the requirements to query the Active Space Middleware Operating System (Gaia in our case, also referred to as meta-OS) to obtain a list of matching entities. Finally, the Controller can include an optional number of mappings for the Adapter (if no mappings are defined, the Adapter simply forwards the requests).

```
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 ListViewer
 Cardinality   1 *
 Requirements
  device=ExecutionNode
    and Type=TouchScreen
    and OS=Windows2000
    or OS=WindowsCE
  Mappings
   selectedEntryChanged =
   playSong
}
Coordinator  {
  ClassName  Coordinator
  Cardinality  1 1
  Requirements
  device=ExecutionNode
     and OS=Windows2000
}
```

```
Application =
{
 Model =
 {{
 ClassName="JukeboxModel",
   Hosts={{ "amr1.as.edu"}},
 }}
 Presentation =
 {{
    ClassName ="MusicPlayer",
    Hosts={{"amr2.as.edu"}}
 }},
 Controller =
 {{
    Classname ="ListViewer",
    Hosts={{"plasma1.as.edu"},
             {"pda1.as.edu"},
           },
    AdapterMappings = {
      {"selectedEntryChanged"
       ,"playSong"},
    }
 }},
Coordinator =
 {{
    ClassName ="Coordinator",
    Hosts={{"amr3.as.edu"}},
 }},
}
```

Fig. 4. Music Jukebox AGD (left). Music Jukebox ACD customized for an active meeting room (right).

The ACD is an application description that customizes an AGD to the resources of a specific Active Space. The ACD consists of information about what specific components to use, how many instances to create, and where to instantiate the components. The Controller component includes the mappings specified in the AGD.

Figure 4 (left) presents the AGD defined for an application called Music Player, which provides functionality to organize and play a collection of music files using resources present in the ubiquitous computing environment. The Model, for example, is implemented by a component named JukeboxModel, has a cardinality of one (a Music Jukebox application has exactly one Model), and requires an ExecutionNode device running Windows 2000. Gaia uses the term Execution Node to abstract any device capable of hosting the execution of Gaia components (e.g., Model, Presentation, Controller, Adapter, and Coordinator). Figure 4 (right) illustrates an ACD customized for a prototype active meting room.

The mapping mechanism receives an AGD and a target Active Space, and generates and ACD customized for such space, according to a mapping 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 4 could be customized to the active meeting room with one to as many song selectors as compatible execution nodes present in the space, 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 mapping mechanism offers two modes of operation: manual and automatic. In the manual mode of operation, users interact with a GUI that parses an application AGD and allows them to drive the mapping process by choosing the devices where the different application components will be instantiated. The automatic mode uses a service called ACDGenerator, which does not require user intervention and uses policies to drive the ACD generation process.

Based on our experience using a prototype Active Space, ACDs are not generated each time an application is started. Instead, ACDs are generated once (when no ACD is available for a specific application and a specific Active Space) and reused later on, as long as the configuration of the Active Space does not change. For example, we often use a Presentation Manager application to present slide-shows. We have a number of default ACDs for this application that allows us to instantiate the application using the displays on the left side of the room, right side of the room, and all available displays (each one using an appropriate touch-screen to instantiate the Controller, located in the appropriate side of the room). When a user selects an application, he or she is presented with a list of default configurations. However, the user is also allowed to create his or her own ACD (which can be saved and reused later).

5. Application Management

This section describes the application management functionality provided by the application framework, including instantiation, adaptation, suspension and resumption, mobility, reliability, and termination. 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. We use policies (e.g., scripts, and services) that leverage the interfaces exported by the application framework services to perform each of the management tasks. Policies allow users and developers to customize each of the application management tasks according to their 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).

5.1 Application Instantiation

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 an instantiation mechanism capable of starting application components in any device present in the Active Space and responsible for assembling the components together.

The application ACD contains information about the components required for the application, their names, initial parameters, and their target execution nodes. The application framework leverages the functionality provided by Gaia OS to instantiate the application components and to assemble them together. There are two default instantiation policies: *strict* and *best-effort*. 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 controllers, and therefore, if some of the presentations or controllers crash it does not affect the usability of the application.

5.2. Application Termination

Terminating an application requires removing all application components from all machines. The application Coordinator's interface provides a method that automatically contacts all application components and terminates them. The Coordinator uses the meta-level information that it stores to locate the appropriate components.

Although the default Coordinator implementation terminates all components, an alternative implementation could disconnect the interactive components from the application (presentations and controllers) and terminate the Model and the Coordinator. This approach keeps the interactive components running (although disconnected from any application) so they can be re-used by another compatible application.

5.3. Application Suspension and Resumption

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, and 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[13], 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 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, and their names. The returned ACD can be used to re-instantiate the application, creating 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, and the space still has the resources the application used (mobile devices might not be present anymore). 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 is the task of a specific instantiation policy. The application framework provides a default policy to suspend and resume an application in the same Active Space.

5.4. Application Reliability

When an application is composed of a collection of distributed components running on multiple machines simultaneously, reliability becomes a key factor. The application must be able to monitor the status of the different components, detect faulty components, and react accordingly. Furthermore, due to the diversity of applications, reliability must be configurable at different granularities such as per-application instance basis or per-application type basis.

Current implementation of the application framework encapsulates the reliability policies in the Coordinator. The default policy detects when an application component stops functioning and automatically detaches it from the application using the Coordinator's interface. However, this policy can be replaced with more sophisticated strategies such as for example, automatically restarting and reassembling the crashing component.

5.5 Application Mobility

The application framework provides support for both inter and intra-space mobility. Intra-space mobility is implemented as a library that interacts with the Middleware Operating System to create and terminate components, and with the Coordinator to attach and detach new and terminated components. For example, moving a Presentation requires creating a new instance of the Presentation, attaching it to the application via the Coordinator, and terminating the original instance. The only difference with duplicating is that the latter does not terminate the original instance.

Inter-space mobility is implemented by a service (Mobility Service) that reuses the application management suspension and resumption methods. The service interacts with the Middleware Operating System to detect people leaving and entering the space. When a user leaves, the service obtains a list of associated applications and suspends them. Then, when the user enters an Active Space, the service resumes the suspended applications. More details about mobility can be found at [14].

6. Addressing the Active Space Application Development Key Issues

This section details how the application framework presented in this paper addresses the issues listed in Section 2. Resource-Awareness (first issue) is addressed by the Gaia Middleware Operating System; the application development middleware services simply leverage the existing functionality (Gaia OS Space Repository) to find resources present in the current environment and relevant to the application. Multi-Device utilization (second issue) is supported by the application model defined by the middleware. The functional decomposition of applications into a Model, a number of presentations, controllers, adapters, and a Coordinator to manage all previous components, simplifies the mapping of different application aspects to different (heterogeneous) devices. Furthermore, implementing each functional unit as a distributed component allows instantiating them in different devices. User-Centrism (third issue) is supported by the intra- and inter-Active Space application mobility functionality provided by the Application Management. Users can move and duplicate components across the Active Space and can move to different Active Spaces and have their applications following them. Run-Time Adaptation (fourth issue) allows controlling the composition of the application dynamically. This functionality is implemented by the application Coordinator (functionality to attach and detach components dynamically) and it is supported by the distributed nature of the Application Model. Application mobility (fifth issue) is directly supported by the Application Management Functionality via the inter- and intra-Active Space mobility protocols. Context-Sensitivity (sixth issue) is supported by the Application Model by means of context Controller. These are controllers that receive context information and trigger changes in the application accordingly. The Controller is the mechanism to introduce context in the application, but it does not provide functionality to synthesize context information from sensors. Instead, it relies on existing services, such as the Gaia OS Context Service. Finally, Active Space Independence is supported by the Application Mapping mechanism, which supports the generation of Active Space customized ACDs .These ACDs allow portability of applications across heterogeneous Active Spaces.

The application framework provided by Gaia meta-OS covers the challenges related to Active Spaces and simplify the development of portable applications. Application developers focus on the functionality related to the application (e.g., playing music or collaboratively editing a document) and leverage the functionality provided by the application framework to supports tasks that are common to most Active Space applications (e.g., mobility, multi-device utilization, and context-awareness).

7. Music Player Example

We present in this section the Music Player Application, an application based on our application framework that provides functionality for playing music files taking benefit of the resources contained in the Active Space where the application is instantiated. The application base-level provides functionality for managing a collection of music files distributed among different devices located in different Active Spaces, allows selecting, controlling, and playing a specific song in the user's current location, and exports information about the play list contents, as well as the currently selected song. The application provides also functionality to register, unregister, duplicate, and move presentations and controllers dynamically, adapts to context changes, and uses mobility policies to follow the user to different Active Spaces.

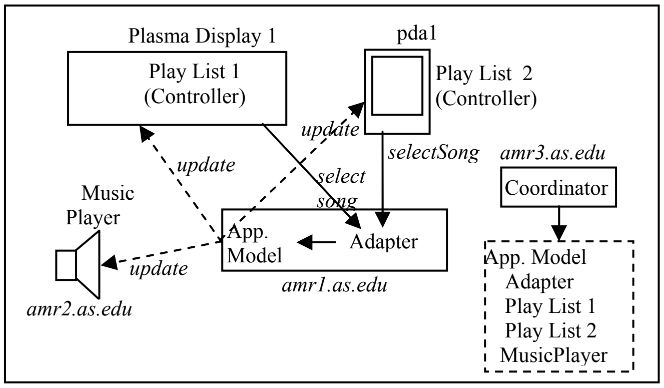

Fig. 5. Music Application Composition.

7.1. Implementation Details

In this example we focus on our active meeting room (Figure 1), managed by Gaia OS. Figure 4 illustrates the AGD for the music application, which consists of a Model, a Presentation (player), a Controller, and a Coordinator. The Coordinator automatically instantiates a default Adapter that maps the events from the Controller (e.g. entry selected) into method requests to the Model (e.g. play). The MusicPlayer Presentation interacts with a commercial-off-the-shelf application to play the audio. The response time of the application is within an acceptable range from an interactive point of view. For example, selecting a song requires less than a second to execute and manipulating the meta-level (duplicating, moving, attaching and detaching presentations and input sensors) takes from 3-6 seconds depending on the request.

7.2. Instantiating and Using the Application

We describe in this section the Music Application's instantiation process. The user enters the active meeting room, registers his or her PDA, and selects a strict instantiation policy to create the application according to the ACD illustrated on the right side of Figure 4. The diagram depicted in Figure 5 illustrates the resulting application partitioning.

When the user selects a song using the PDA's Controller, this sends an event (selectedEntryChanged) to the Adapter with the name of the song. The Adapter sends a request to the Model (playSong), which sends an update to the music player Presentation, and to the two controllers (List Viewers). The player gets the music data from the Model and starts playing, and the list viewers get the name of the currently selected song and highlight the name in their list.

8. Performance Evaluation

The main goal of the application framework is to provide support for the construction of a new type of applications we refer to as Active Space applications. In order to evaluate the framework, we have focused on whether or not the functionality provided is sufficient, rather than performance. Both Gaia OS and the application framework are built on top of Orbacus, which is an efficient and fast CORBA implementation, and CORBA UIC[15], a customized and efficient minimalist CORBA ORB. Therefore, the response time of the system is well within an acceptable interactive response time and comparable to interactive desktop applications.

In order to evaluate the application framework, we have built fourteen applications that have allowed us to validate the framework. The fourteen applications show that the application framework is generic enough to cover a large range of interactive Active Space applications.

We present in this section a performance evaluation for a slideshow application we use regularly in our Active Space (Presentation Manager). The application consists of a Model that keeps information about the state of the slideshow (e.g., slideshow name, slideshow file's path, and current slide), a Presentation that uses Microsoft Power Point to render the slides (via the COM interface), and a VCR Controller with functionality to start and stop the slideshow and navigate the slides. The application allows presenting synchronized slides in multiple displays simultaneously, and can also have multiple VCR controllers attached simultaneously. Furthermore, it provides functionality for intra- and inter-space mobility (default application framework functionality). We present next, a performance evaluation for application instantiation, moving a Presentation (slide viewer) from one display to another (intra-space mobility), navigating slides, and terminating the application. All the tests were performed in our prototype Active Space, which has a 1Gb Ethernet network, 802.11b, 15 Pentium IV at 1.2 GHz with 256MB of RAM, and 4 61" Plasma displays. All the times presented are the average result of ten experiments.

Figure 6 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. Each configuration corresponds to a different ACD. The time was calculated from the time we start the application until the first slide is displayed by all presentations. The average time increases linearly as the number of presentations increases. The time required to start Microsoft PowerPoint in one machine by double-clicking the icon and starting the slideshow is 0.85 seconds (no Gaia OS or application framework). Starting the Presentation Manager with one Presentation and one VCR Controller takes 2.18 seconds, while the same application without the VCR Controller requires 1.13s. These times include creating the Model, the Coordinator, a Presentation, and one or zero controllers, and assembling them together using the Coordinator interface. All components except the VCR Controller are implemented as DLLs and creating them requires loading them in a pre-created process (Component Container). The VCR Controller, on the other hand, is an executable. Creating a new executable takes longer than loading a DLL (at least in Windows), which explains the 1.05 additional seconds required to instantiate the application with the Controller. Based on the previous results, the impact of the application framework is negligible. According to Figure 6, there is a penalty of approximately 1s for each additional Presentation. This number is the time required to create the PowerPoint COM object plus the time required by this object to render the first slide (the Presentation creates the COM object and sends requests to display slides). It is possible to improve the instantiation time. Our current instantiation policy instantiates all presentations sequentially, and 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 asynchronous 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 Gaia OS Space Repository). In this case, the time would be significantly smaller, regardless the number of presentations because all presentations would be instantiated in parallel.

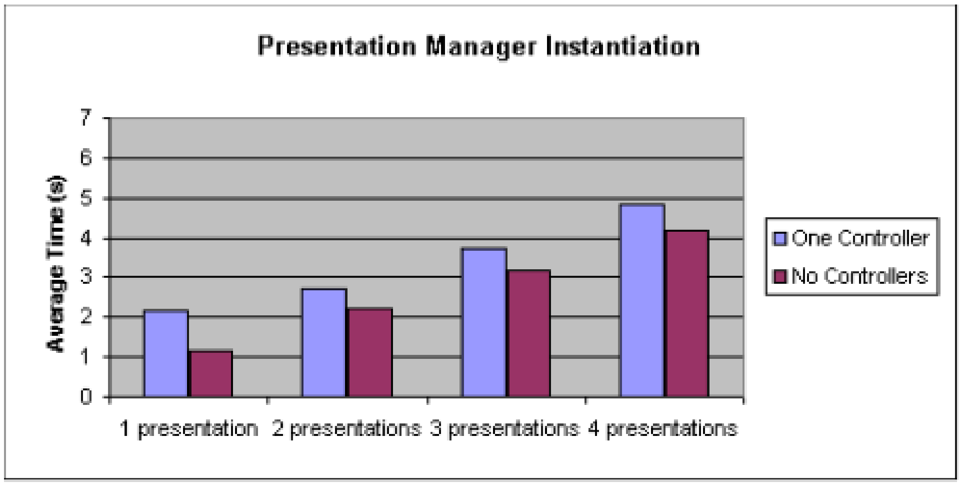

Fig. 6. Average time to instantiate the Presentation Manager application.

Next experiment calculated the time required to move a Presentation (slide) from one display to another. This time included creating a Presentation in the execution node associated to the target display, attaching it to the Coordinator, unregistering and terminating the original Presentation, and finally the time required by the new Presentation to display the current slide (the new Presentation gets the current state by interacting with the Model). The average time based on ten experiments was 2 seconds.

Based on our experience with all the applications, the interactive application response time is similar to a desktop application. For example, in the case of the Presentation Manager, the time it takes to move to the next or previous slide since we press a button in a VCR Controller (running on a wireless connected PDA or on a wired connected touch screen) is the same as in a standard Power Point application running on a PC (e.g., pressing the space bar), which is on average below a second. This time includes sending an RPC request over the network from the VCR Controller to the Adapter, the Adapter mapping the request to the appropriate method request for the Model, sending an RPC to the Model, the Model updating the current slide number and sending a notification (asynchronous RPC) to the presentations (the notification includes the slide number), and the presentations parsing the notification and rendering the appropriate slide via the PowerPoint COM object. Presentations cache the slideshow file locally at the beginning of the slideshow so they only ask for the file once (they obtain the file from the Gaia Context File System). The time is bounded by the Power Point rendering engine, not by the mechanisms implemented by the application framework

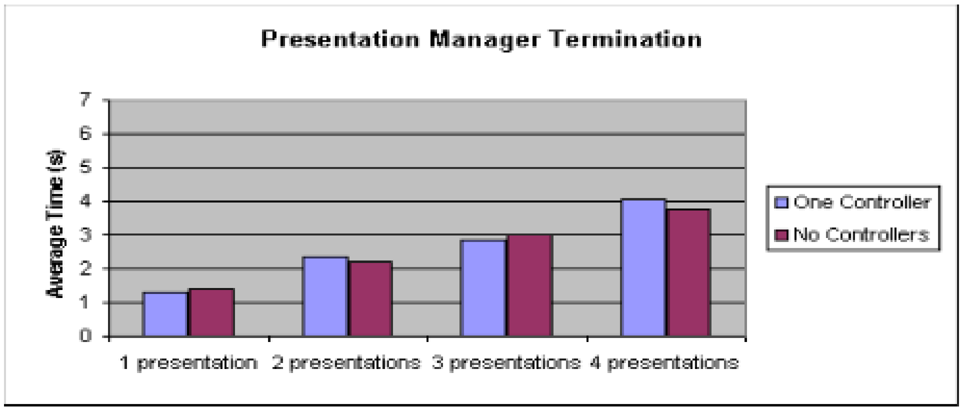

Fig. 7. Average time to terminate the Presentation Manager

Our next performance evaluation calculates the time required for terminating the application. The Coordinator exports a method that implements this functionality. The method interacts with the Model, Presentation(s), Controller(s), and Adapter(s), notifies them that they are being unregistered from the application (the components can then implement cleaning-up procedures including resource release), and uses the Gaia Component Management Core functionality to terminate all components, including itself. In the Presentation Manager application, the only components that implement clean-up functionality are the Presentations. When they receive the notification, they stop rendering the slide and terminate the execution of the PowerPoint COM object. For our experiment, we calculated the execution time of the Coordinator's *terminateApplication* method. We used the same configurations as in the instantiation experiments, that is, one, two, three, and four presentations, once with a VCR Controller, and the second time without the VCR Controller. Figure 7 illustrates the termination times. In this case, the average time for terminating an application with or without a VCRController is roughly the same (the time required by Windows to terminate the VCR Controller executable is negligible).

Finally, suspending and resuming an application is similar to terminating and instantiating an application respectively, with additional required time to save the state (suspend) and restore the state (resumption). We have performed some experiments suspending and resuming Presentation Manager in the same Active Space (we reuse the same ACD). The time to save the state stored by the model and the coordinator is on average 30ms (using Gaia's distributed file system), while the time to restore the state took, on average, 50ms. Therefore, the time required to suspend and resume an application is bounded by the termination and instantiation times.

Based on the performance evaluation and on our experience with the rest of Gaia applications, the Application Framework does not introduce any overhead on the overall application response time, compared to most traditional desktop applications.

9. Related Work

The Pebbles [2] project is investigating partitioning user interfaces among a collection of devices. Pebbles is mostly concerned with issues related to GUIs, and the proposed infrastructure does not provide functionality for dynamically altering the partitioning layout. Our application framework focuses on the application composition, management, adaptability and configurability, and provides reflective functionality that allows altering the application structure at run-time.

BEACH [16] 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.

Graspable Interfaces [17] presents 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. Our framework combines both concepts and defines the time-space-multiplexed model.

The PIMA [18] and I-Crafter [19] projects propose a model for building platform independent applications. Developers define an abstract application that is automatically customized at run-time to particular devices. PIMA and I-Crafter generate applications for a single device, while we consider applications partitioned across devices. However, we can leverage the functionality provided by both approaches to dynamically generate application presentations customized to specific devices.

The Presentation-Abstraction-Controller[20] (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 distributed across View-Controller pairs. The Abstraction-Link-View[21] (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 described in this paper, although reusing the original concepts from MVC, uses techniques present in PAC and ALV.

Projects such as Stanford's iROS [22] and CMU's Aura [23] provide a middleware infrastructure to manage ubiquitous computing environments. However, none of them provides an explicit middleware infrastructure customized to support application development.

10. Conclusions and Future Work

This paper presents our application framework for designing and building user-centric, resource-aware, context-sensitive, multi-device, mobile applications. These applications are bound to users instead of devices, can take benefit of resources present in the users' environment, can react to changes in the environment, and can be partitioned among different devices.

The application framework defines an application model that provides a component (Coordinator) to access and modify the composition of the application dynamically, implements a mechanism to define applications abstractly and manually or automatically map them to arbitrary environments, uses flexible policies to separate the basic application construction and modification functionality from particular strategies.

We have successfully implemented the functionality described for Gaia OS and the application framework, and have fourteen applications that prove that the framework simplifies the design and implementation process. Furthermore, the flexibility and dynamism of such applications has simplified the interaction with Active Spaces such as our prototype active meeting room. The framework allows integrating existing components including Microsoft COM objects (e.g., Power Point) as presentations, controllers, and models, and extends the functionality of these components by allowing users to move the component across different devices, and even extend them for collaborative environments. Integrating existing components is done by having a Presentation, Controller, or Model wrapping the existing components and delegating the application framework-related requests to the wrapped component.

Although we have not fully reached the proposed customizable habitat vision yet, we believe that the application framework presented in this paper is a valid solution to program existing device rich environments.

11. Acknowledgements

This research is supported by the National Science Foundation grant NSF 98-70736, NSF 9970139, and NSF infrastructure grant NSF EIA 99-72884

References

1. Roman M, Hess CK, Cerqueira R, Ranganat A, Campbell RH, Nahrstedt K: Gaia: A Middleware Infrastructure to Enable Active Spaces. IEEE Pervasive 1:74-82, 2002
2. Myers BA: Using Hand-Held Devices and PCs Together, Communications of the ACM, vol 44, 2001, pp 34-41
3. Abowd GD, Mynatt ED: Charting Past, Present, and Future Research in Ubiquitous Computing. ACM Transactions on Computer-Human Interaction 7:29-58, 2000
4. Costa FM, Blair GS, Coulson G: Experiments with an architecture for reflective middleware. IOS Press 7:313-325, 2000
5. Kon F, Singhai A, Campbell RH, Carvalho D, Moore R, Ballesteros FJ: 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments. Paper presented at the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, Brussels, Belgium, July 1998 1998
6. Kiczales G, Rivires Jd, Bobrow DG: The Art of the Metaobject Protocol. MIT Press, 1991
7. Kiczales G: Beyond the Black Box: Open Implementation. IEEE Software 13:137-142, 1996
8. Blair G, Coulson G, Robin P, Papathomas M: An Architecture For Next Generation Middleware. Paper presented at the IFIP International Conference on Distributed Systems, Platforms, and Open Distributed Processing, Lake District, England, September 1998
9. Dey AK: Providing Architectural Support for Building Context-Aware Applications, PhD Thesis in Computer Science. Atlanta, Georgia Institute of Technology, 2000, pp 188
10. Krasner GE, Pope ST: A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. Journal of Object Oriented Programming 1:26-49, 1988
11. Coutaz J: PAC: An object-oriented model for dialog design. Paper presented at the INTERACT'87: The IFIP Conference on Human Computer Interaction, Stuttgart, Germany, 1987
12. Salber D, Dey AK, Abowd GD: The Context Toolkit: Aiding the Development of Context-Enabled Applications. Paper presented at the CHI'99, Pittsburgh, May 1999
13. Hess C, Campbell RH: A Context-Aware Data Management System for Ubiquitous Computing Applications. Paper presented at the International Conference in Distributed Computing Systems (ICDCS 2003), Providence, Rhode Island, May 19-22, 2003 2003
14. Roman M, Ho H, Campbell RH: Application Mobility in Active Spaces. Paper presented at the 1st International Conference on Mobile and Ubiquitous Multimedia, Oulu, Finland, 2002
15. Roman M, Kon F, Campbell RH: Reflective Middleware: From Your Desktop to Your Hand. IEEE Distributed Systems Online. Special Issue on Reflective Middleware, 2001
16. Tandler P: Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices. Paper presented at the Ubicomp 2001: Ubiquitous Computing, Atlanta, Georgia, September 30 - October 2 2001
17. Fitzmaurice GW: Graspable User Interfaces, PhD Thesis in Computer Science. Toronto, University of Toronto, 1996
18. Banavar G, Beck J, Gluzberg E, Munson J, Sussman JB, Zukowski D: An Application Model for Pervasive Computing. Paper presented at the 6th ACM MOBICOM, Boston, MA, 2000
19. Ponekanti SR, Lee B, Fox A, Hanrahan P, Winograd T: ICrafter: A Service Framework for Ubiquitous Computing Environments. Paper presented at the Ubicomp 2001: Ubiquitous Computing, Atlanta, Georgia, September 30 - October 2 2001
20. Coutaz J: PAC, an Object Oriented Model for Dialog Design. Paper presented at the Human Computer Interaction. INTERACT 1987 1987

21. Hill RD: The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interface to Applications. Paper presented at the CHI May 3-7 1992
22. Johanson B, Fox A, Winograd T: Experiences with Ubiquitous Computing Rooms. IEEE Pervasive Computing Magazine 1:67-74, 2002
23. Sousa JP, Garlan D: Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. Paper presented at the IEEE/IFIP Conference on Software Architecture, Montreal, August 25-31 2002