APPLICATION MOBILITY IN ACTIVE SPACES

Manuel Román, Herbert Ho, and Roy H. Campbell

{mroman1, hho, rhc}@cs.uiuc.edu
University of Illinois at Urbana-Champaign
http://www.cs.uiuc.edu

Abstract. The vision of Ubiquitous Computing is that of users naturally interacting with their environment to access digital data and applications. The digital world is finally merged with the physical world and users can interact with it to perform a number of tasks. However, one of the requirements of this vision is the ability to build and run applications that are not bound to a specific device. These applications are associated with a user and have the ability to move with the user and adapt automatically to different environments.

We present in this paper an application framework and an associated system software infrastructure to support the development of mobile applications. We provide two types of mobility support: inter-space mobility to move applications across different spaces, and intra-space mobility to move the components of an application to different resources present in the user's current environment.

1. Introduction

Mark Weiser, in his seminal paper "The Computer for the 21st Century" [22], defines ubiquitous computing as a technology that "resides in the human world and weaves itself into the fabric of everyday life". We use the term active space to refer to an enhanced user habitat that integrates the physical properties of the space with its digital counterpart. An active space is capable of sensing the context of the habitat, can react accordingly, and allows users to customize it according to their needs. The computational focus is on the active space (habitat) and not on individual computers or devices.

It is possible to provide a system software infrastructure to activate arbitrary physical spaces (e.g. house, office, meeting room, car, and bus station). We refer to the software infrastructure that abstracts the active space as a programmable computing environment as a meta-operating system [12]. Active spaces coexist with other active spaces. An active house is part of an active city and it is composed of active rooms. From this perspective, ubiquitous computing is not an isolated experience but a continuous presence that enhances our ability to access a greater amount of information and gives us the means to configure our environment.

The traditional application-to-computer association disappears in favor of an application-to-active space association. This association is temporal and applications are migrated to different active spaces as users move from on active space to another. Users can roam across different active spaces and have their applications move with them seamlessly (inter-active space mobility).

In this paper, we address the issue of application mobility in active spaces and present the results of our experiments with applications in a prototype active space illustrated in Figure 1. This active space is coordinated by a meta-operating system called Gaia OS [17] we have developed, and the applications are built using the application framework it provides. Unlike traditional PC applications, an active space application may be started in one active space and moved to another active space. Furthermore, an active space application may require dynamic adaptation to move different elements of the application to different devices in the space (intra-active space mobility). For example, a slide show application may start with two displays and as new people enter the room it may use new displays, or may move the slides to a location that is visible by all people in the room. Furthermore, if the people move to a bigger room the application must be able to move the new space and continue with the presentation with minimal or no user intervention. These applications are subject to a new set of assumptions [16] and require new supporting infrastructures.

We present the Gaia OS meta-operating system in Section 2 and the associated application framework in section 3. Section 4 focuses on intra-space mobility and section 5 presents inter-space mobility, including a classification (section 5.1), and the support for two mobility types (sections 5.2 and 5.3). We present the implementation details in section 6, talk about related work in section 7, and conclude the paper and describe our future work in section 8.

2. Supporting Meta-Operating System: Gaia OS

Gaia OS provides services for location, context, events, and repositories with information about the active space.

Figure 1. Prototype Active Space.

The system is built as a distributed object system. Figure 2 shows the three major building blocks of Gaia: the Gaia Kernel, the Gaia Application Framework, and the Applications.

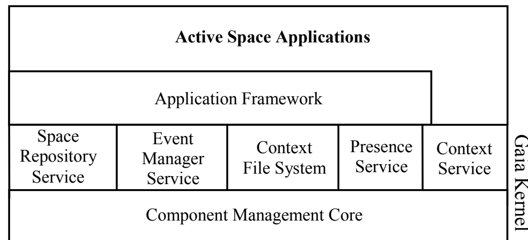

Figure 2. Gaia OS Infrastructure.

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 Kernel's five basic services are the Event Manager Service, Presence Service, Context Service, Space Repository Service, and Context File System.

Gaia applications use a set of component building blocks, organized as the Gaia Application Framework[16], 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. The Active Space Application layer contains applications and provides functionality to register, manage, and control these applications through the Gaia Kernel services.

The *event service* is responsible for event distribution in the active space and implements a decoupled communication model based on suppliers, consumers, and channels. The *context service* allows applications to query and register for particular context information so that they may adapt to their environment. The *presence service* is responsible for detecting digital and physical entities present in an active space. It implements a beaconing mechanism to maintain soft-state about digital entities present in the space, and uses different types of sensors to proactively detect the presence of physical entities. 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 *context file system* [9] incorporates context into the traditional file system model to provide support for mobile users, device heterogeneity, and data organization. Every active space has a file system instance that aggregates data from different sources.

3. Application Framework for Active Spaces

This section provides a brief description of an application framework customized for active spaces that successfully addresses inter-active space mobility as well as intra-active space mobility (moving pieces of an application inside an active-space).

Active space applications separate application semantics from application composition. The number and type of resources present in an active space implies that there are several valid application composition configurations. However, regardless of the different compositions, the semantics of the application remain constant – the goal of a music player application is to play music, regardless of how many presentations and input sensors we attach. The application framework we present in this section uses computation reflection[19] to explicitly separate the application base-level (application domain functionality) from the application meta-level (application composition); resulting applications are considered open implementations[10]. Computational reflection is an effective mechanism to manage the complexity involved in the development of active space applications, allowing developers to concentrate on the application base-level and providing mechanisms to automate the application meta-level configuration dynamically. Computational reflection has been successfully applied to in the past to middleware infrastructures [3, 6, 7, 11] to allow users reconfigure the middleware services dynamically, while keeping applications unchanged.

The application framework we propose models applications as a collection of distributed components and reuses the application partitioning proposed by the Model-View-Controller[13]. 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), and is context-sensitive.

The framework is divided into three main building blocks: infrastructure, mapping mechanism, and management interface.

3.1 Infrastructure

The application framework defines five components (Figure 3): model, presentation (generalization of View), controller, input sensor, and coordinator. The model, presentation, controller, and input sensor 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.

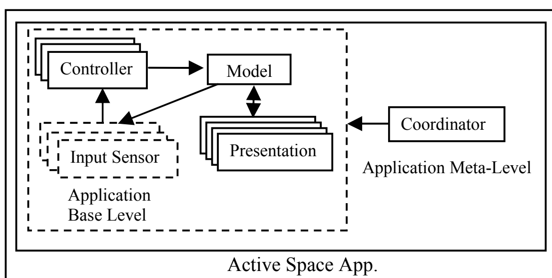

Figure 3. Application Framework Infrastructure.

The model implements the logic of the application and exports an interface to access and manage the application's state. The model is the only base-level component in the application that maintains state. The model maintains a list of registered listeners and it is responsible for notifying them about changes in the application's state, therefore keeping them synchronized.

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. Presentations are dynamically attached and detached to and from the model. When a presentation receives an update event from the model, it contacts the model and retrieves the new application state.

We use the term input sensor to refer to any entity (i.e., hardware and software) capable of generating events that can alter the application's state. Examples of hardware input sensors are mice, keyboards, active badges, i-buttons, and fingerprint detectors. Examples of software input sensors are GUIs containing widgets that can be associated to user-defined events, and context input sensors, which are entities that process different context properties and synthesize specific context events.

The controller coordinates the interaction between input sensors and the model. It maps events generated by input sensors into method requests implemented by the application model, therefore decoupling input sensors from particular applications. The mappings can be added and removed dynamically.

Active space applications are a collection of distributed components composed of a model, a controller, and a number of presentations and input sensors. 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 in a single process) and therefore those issues are not a concern. For an active space application, the number and location of presentations and input sensors 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 input sensors, 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 and inputs sensors. The coordinator also provides functionality to retrieve run-time information about the composition of the application components, as well as functionality for fault-tolerance and application lifetime management. This functionality offers fine-grained control over the application internal composition rules.

3.2 Application Mapping

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

The application framework allows applications to be built that are independent of particular active spaces. The framework defines two types of application descriptions: the application generic description (AGD), and the application customized description (ACD).

The AGD is an active space-independent application description that lists the components of an application and their requirements. The list includes one model, one coordinator, zero or more presentations, and zero or more input sensors. 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 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 equivalent to an executable file in a traditional operating system. They provide a detailed description of the application components, which the instantiation mechanism uses to start the application.

The mapping mechanism receives an AGD and a target active space, and generates an ACD customized for such space. The diversity of resources present in an active space allows for multiple application configurations (ACDs). The mapping mechanism provides two modes of operation: manual and automatic. In manual mode, users interact with a graphical tool to drive the mapping process. The automatic mechanism uses a service that generates the ACD without user intervention, and uses configurable policies to drive the mapping. These policies are user defined, and can range from simple template based approaches (e.g. instantiate presentations in all compatible devices) to artificial intelligence based algorithms (e.g. instantiate presentations in the most appropriate devices according to user location).

3.3 Management Interface

The management interface provides functionality to control applications life-cycle (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. We use policies (e.g., scripts and services) that leverage the interfaces exported by the application framework to perform each of the management tasks. 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.

4. Intra-Space Mobility

Intra-space mobility allows moving the interactive components of an application (i.e. presentations and input sensors) 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 intra-space 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 4 illustrates a schematic of the intra-space mobility functionality.

Moving a presentation or an input sensor (we use a presentation as an example) starts by obtaining compatible devices for the selected component (1).

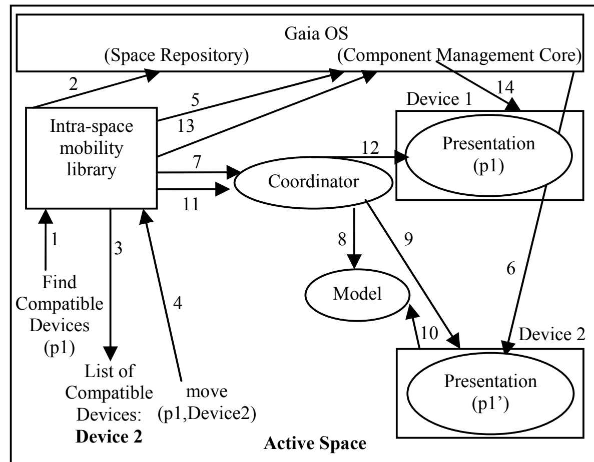

Figure 4. Intra-active space mobility functionality schematic.

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 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(4). This method contacts the Gaia OS Component Management Core (CMC) (5) to create the component (p1') 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 as a listener (8), and notifies the presentation it has been attached to an application (9). The presentation contacts the model (10) 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 (11). The coordinator notifies the presentation it has been detached from the

application (12), therefore allowing the presentation to release resources. Finally, the library uses the Gaia CMC (13) to delete the original presentation (14).

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

5. 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 through different 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.

5.1 Mobility Classification

Our current inter-space mobility implementation defines two types of mobility: interactive and full. Interactive mobility means that only interactive components of the application move (i.e., presentations and input sensors), while full mobility implies that all the components of the application move to a new space.

There are, at least, two situations where interactive mobility is useful: the task implemented by an application is associated to a particular active space, or the application is being used by different users simultaneously (or both).

As an example of an application associated to an active space, consider a video monitoring application. The user is interested in monitoring a specific active space. When the user moves, he or she wants to be able to see the video using a presentation component, but still wants to monitor the same space. In this situation, the mobility policy moves the presentation but leaves the model, coordinator, and additional interactive components in the same active space. We call this mobility pattern *selective interactive*. Users can interact with the mobility service to define what interactive components to move.

The application framework supports the construction of collaborative applications. Every application has an owner, and different users can attach presentations and input sensors to use the application. Input sensors and presentations have also an owner, which may be the same or different from the user who created the application. For example, we have a presentation manager application that allows managing PowerPoint slideshows. Most of the times, the speaker starts the application (he or she becomes the owner), and other people register presentations and input sensors to see and edit the slides. In this situation, if one of the user moves to another active space (e.g., his or her active office) it would not be appropriate to move the whole application to the new space. The interactive mobility policy provides functionality to move all presentations and input sensors owned by the user to the new active space. We refer to this interactive mobility policy as *full interactive*. Based on the presentation manager example, a user can move to a different active space and have the presentation and input sensor following him or her. In this way, he or she can still remotely participate in the presentation.

Full mobility provides functionality to move all the components of an application to a new active space and is mostly used with user-centric applications. These applications provide functionality that is customized for a user, and therefore, are not bound to a specific active space. As a result, the applications move with the user, have access to the user's data regardless of the user's location, and benefit from the resources present in the user's surroundings, adapting their structure if required. Examples of these applications are personal information management applications (e.g., calendar, to-do list, and address book), music players, document readers, word processors, or in general, any application whose lifespan is not bound to an active space. Full mobility supports the concept of "everyday computing" defined by Abowd et al. [1], which refers to applications without a clear end that are associated with users, and therefore require mobility capabilities.

5.2 Interactive Mobility Implementation

Interactive mobility defines that the model and the coordinator remain in the same active space and only interactive components move. The implementation of this type of mobility is similar to the mechanisms described for intra-space mobility. Selected interactive components are instantiated in the new active space, and reconnected to the application using the original coordinator.

The mobility service leverages configurable policies to determine the most appropriate devices to instantiate the components (e.g., previous usage patterns, artificial intelligence, or user intervention), and uses persistent storage to store the references to the original coordinator. This information is saved in the user personal profile (always accessible) when the user leaves an active space.

5.3 Full Mobility Implementation

In this section we describe the mechanisms used to move a whole application across active spaces.

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 functionality relies on the application mapping mechanism to adapt the application to the new environment.

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.

5.3.1 Suspending the Application

Suspending an application involves saving the state of the application and terminating the execution of all the components of the application. The application state comprises base-level state (state managed by the model) and meta-level state (state managed by the coordinator). Saving the base-level state is application dependent and requires intervention of the application model, which must implement the *saveState* method. For example, in a music application, the base-level state includes the play list, the name of the song currently playing, and the current playing time. The state information managed by the coordinator includes number of components, types, and the execution nodes where these components run. This information is independent of the application base-level functionality. The coordinator provides functionality to generate a "snapshot" of the application composition as an ACD, which contains a list of the application components and the execution nodes that host their execution. This ACD can be used to start a clone of the original application.

The mobility functionality relies on the Gaia Context File System (CFS) [9] to store the application state. The destination of this state is configurable and could be the user's home active space, or the user's PDA, which is registered with the space and mounts its local file system as part of the active space file system. The application suspension functionality is depicted in figure 5.

Initially the inter-space mobility service receives a notification about a user leaving the active-space (1). The service uses the name of the user to obtain a list of associated applications from the space repository and suspends all the applications (users can configure what applications must be moved). For each application, the service sends a request to the coordinator to save the state (2), which the coordinator forwards to the model (3). The model serializes the application state and uses the CFS path passed as a parameter to save the state (4). Next, the mobility service sends a request to the coordinator to generate an ACD (5) and saves it in the CFS path passed as a parameter (6). This ACD is used later during the application resumption process. Finally, the mobility service sends a request to the coordinator to terminate the application (7). The coordinator contacts all execution nodes hosting application components and destroys them.

5.3.2 Resuming the Application

Resuming traditional applications is implemented by obtaining the serialized information, restarting the application, and using the serialized data to bring the application to the original state. Most existing mechanisms [8, 14, 20] assume that the structure of the application does not change.

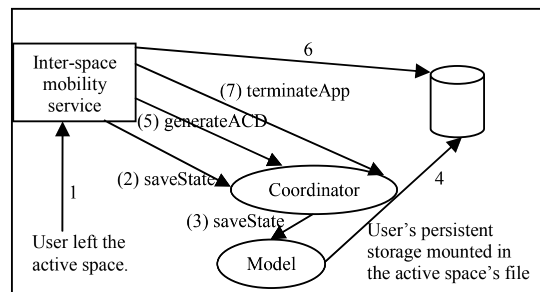

Figure 5. Suspending an application.

However, in an active space, this 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 application resumption management interface takes into account the issues mentioned above, and provides functionality to resume a previously suspended active space application. Figure 6 illustrates the steps required to resume an application.

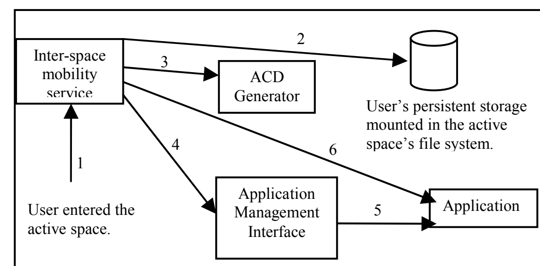

Figure 6. Resuming an application.

When a user enters an active space, the Gaia Presence Service detects it and sends a notification that is received by the Inter-Space Mobility Service (1). The active space mounts the user persistent storage in the space's file system hierarchy. 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 space repository 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 saved 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 their requirements, 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 space repository, the ACD Generator generates a new ACD and instantiates the application.

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.

Resuming an application in a new active space may not always be possible. The target active space may not have appropriate resources, may not allow application instantiation due to security restrictions, or may have all resources used by other applications. In this situation, the mobility service notifies the user about the problem and stops the mobility process.

After we obtain a valid ACD, the mobility service uses the application management interface (4) to instantiate the application using the ACD (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. Implementation Details

The Gaia OS meta-operating system kernel services run on Windows 2000/XP. The services are written in C++ and Java, and use CORBA as the default middleware implementation. We also have some Gaia components (Execution Node and application framework components) running on Windows Pocket PC devices and using CORBA to interact with the active space services. The CORBA implementation for Windows 2000/XP is Orbacus, while the implementation for the handheld devices is UIC[18]. The default scripting language in Gaia OS is LuaORB [4], which provides functionality to interface CORBA objects easily. We currently use Lua to implement the Gaia OS bootstrap and most of the system policies (e.g. instantiation, ACD Generator policies, and active space mobility).

We have implemented nine applications based on the proposed application framework. All applications have built-in functionality for intra-active space mobility, and two of them (Music Player and Presentation Manager) implement the *saveState* and *restoreState* methods to support inter-space mobility. In this section, we provide details about the Music Player application.

The Music Player application provides functionality to play music in different formats (e.g., mp3, wav, and wma) and consists of a model, a play list input sensor (displays a list of available songs that users can select), a music player presentation that re-uses a COTS application, and a coordinator. We reuse the default coordinator provided by the application framework and implement the rest. The model stores information about the play list, the currently selected song, the current play time, implements the *saveState* and *restoreState* methods, and provides an interface to manage and obtain the play list, get the current playing time, and start and stop the song. The music player presentation receives notifications from the model and controls the music player component. Finally, the play list input sensor displays a list of available songs (songs can be added and removed at run-time) and allows users to play a song by selecting one of the entries.

The model and presentation are implemented so the music is synchronized when using intra and inter-space mobility. For intra-active space mobility, when the music presentation is detached (it is being moved), it sends a request to the model to pause the audio. When the new presentation (in

the target device) is attached, it contacts the model, obtains the song file (caches it locally), obtains the stored playing time (when the song was paused), and resumes the audio at that point. For inter-space mobility, when the application is suspended, the model stores information about the play list, the current song, and the current time. When the application is restarted, the model uses the stored information to obtain the play list, selects the previously saved song, and sets the current time to the stored value. When the new presentation is attached, it contacts the model, gets the current song, the current time, and starts playing.

7. Related Work

The PIMA [2] and I-Crafter [15] projects propose a model for building portable platform independent applications. Developers define an abstract application that is automatically customized at run-time to particular devices. The mapping mechanism implemented by the Gaia Application Framework is based on the same idea. The difference is that Gaia maps an application to an active space, while PIMA and iCrafter map applications to a single device. Furthermore, the Gaia Management Interface provides support for application mobility, which is not present in any of the two previous approaches.

BEACH [21] 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 both inter and intra-space mobility. BEACH provides support for intra-space mobility but they do not provide support for inter-space mobility, because they target a single scenario.

The iMASH project provides "session handoff mechanisms" and identifies three different kinds of session handoff[14]. It provides an optimization mechanism to cache and transform data so the system does not need to transmit all data from client to client. iMASH and Gaia goals are different. Gaia targets multi-device applications and the inter-space functionality supports moving applications from space to space (collection of devices), not from device to device.

Application mobility under Aura [20] 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.

one.world [8] provides support for application mobility (migration) through a hybrid of process migration and application-dependent mobility. 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. 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.

The MIT's MetaGlue [5] is an agent-based system based on Java that supports the development of software for active spaces' environments. The system provides support for resource management, event broadcasting, real-time response, saving and restoring the agents' state, and distributed debugging. The system supports agent migration and agent dependency specification, which allows dealing with groups of agents at once. The main difference with Gaia is that the latter provides explicit support for application development and mobility by providing a framework and related mechanisms. Furthermore, Gaia allows migrating applications across different spaces and provides mechanisms to adapt the application the new environment.

8. Conclusions and Future Work

This paper presents an application framework and supporting meta-operating system to assist in the development of mobile ubiquitous applications. These applications are composed of a collection of distributed components, which can be duplicated and moved to different resources contained in an

active space. Furthermore, applications can also move across different spaces and the service providing such functionality is able to negotiate with the different active spaces to find a compatible application configuration. The supporting infrastructure associates applications to users so it knows what applications to move, as well as what are the user preferences for each application. The service can be configured to work with or without user intervention and uses flexible policies to customize different functional aspects.

To our knowledge, this is the first implementation that uses a mapping mechanism to customize an application to the resources present in different active spaces (resource-awareness).

The inter-space mobility service presented in this paper is our first prototype, which has been useful to understand the scope of the problem and the different alternatives. We believe that the functionality is useful not only for mobility, but also to create application sessions or virtual environments. Users can create sessions with a number of applications each (e.g., movie session, presentation session, and office session) and activate them in different spaces at different times preserving the state of the applications. Users can have a "virtual office" that can be mapped in an active office and in an active home. The concept of multiple sessions is similar to the ALT+TAB key in Microsoft Windows, and would allow users to change the behavior of the space simply by changing the active session.

From our experience, one of the most challenging tasks is to obtain a "suitable" ACD when the application is resumed. The number and type of resources present in an active space imply a number of alternative ACDs. While manual selection or generation can be valid in some situations, it is also a tedious task. Users may be annoyed if they have to spend time configuring the applications each time they enter a space. Automatic generation solves this problem but requires appropriate algorithms (most likely AI based) to find a valid configuration. We believe that the combination of both techniques (manual and automatic) and saving previously used configurations is probably the way to go. A user is likely to use the music application in the same way in his or her active house. Our future work includes studying these issues in detail, as well as using semantic information to give meaning to the resources present in the space. This would assist during the ACD generation process.

9. Acknowledgements

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

10. References

[1] Abowd G D and Mynatt E D (2000) Charting past, present, and future research in ubiquitous computing.ACM Transactions on Computer-Human Interaction 7:29-58

[2] Banavar G, Beck J, Gluzberg E, Munson J, Sussman J B and Zukowski D (2000) An application model for pervasive computing. Proc. 6th ACM MOBICOM, Boston, MA, 266-274

[3] Blair G, Coulson G, Robin P and Papathomas M (1998) An architecture for next generation middleware. Proc. IFIP International Conference on Distributed Systems, Platforms, and Open Distributed Processing, Lake District, England,

[4] Cerqueira R, Cassino C and Ierusalimschy R (1999) Dynamic component gluing across different componentware systems. Proc. International Symposium on Distributed Objects and Applications (DOA'99), Edinburgh,362-371

[5] Coen M, Phillips B, Warshawsky N, Weisman L, Peters S and Finin P (1999) Meeting the computational needs of intelligent environments: The metaglue system. Proc. MANSE'99, Dublin, Ireland,

[6] Costa F M, Blair G S and Coulson G (2000) Experiments with an architecture for reflective middleware.IOS Press 7:313-325

[7] Coulson G, Blair G, Davies N, Robin P and Fitzpatrick T (1999) Supporting mobile multimedia applications through adaptive middleware.IEEE Journal on selected areas in Communications 17:1651-1659

[8] Grimm R, Davis J, Lemar E, McBeath A, Swanson S, Gribble S, Anderson T, Bershad B, Borriello G and Wetherall D (2001) Programming for pervasive computing environments University of Washington, Technical Report: UW-CSE-01-06-01

[9] Hess C K (2002) A context file system for ubiquitous computing environments University of Illinois at Urbana-Champaign UIUCDCS-R-2002-2285 UILU-ENG-2002-1729

[10] Kiczales G (1996) Beyond the black box: Open implementation.IEEE Software 13:137-142

[11] Kon F, Costa F, Blair G and Campbell R H (2002) The case for reflective middleware.Communications of the ACM 45:33-38

[12] Kon F, Singhai A, Campbell R H, Carvalho D, Moore R and Ballesteros F J (1998) 2k: A reflective, component-based operating system for rapidly changing environments. Proc. ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, Brussels, Belgium,388-389

[13] Krasner G E and Pope S T (1988) A description of the model-view-controller user

interface paradigm in the smalltalk-80 system.Journal of Object Oriented Programming 1:26-49

[14] Phan T, Guy R, Gu J and Bagrodia R (2001) A new twist on mobile computing: Two-way interactive session transfer. Proc. Workshop on Internet Applications (WIAPP 2001), San Jose, CA,2-12

[15] Ponekanti S R, Lee B, Fox A, Hanrahan P and Winograd T (2001) Icrafter: A service framework for ubiquitous computing environments. Proc. Ubicomp 2001: Ubiquitous Computing, Atlanta, Georgia,56-75

[16] Roman M and Campbell R H (2002) A user-centric, resource-aware, context-sensitive, multi-device application framework for ubiquitous computing environments University of Illinois at Urbana-Champaign UIUCDCS-R-2002-2284 UILU-ENG-2002-1728

[17] Roman M, Hess C K, Cerqueira R, Ranganat A, Campbell R H and Nahrstedt K (2002) Gaia: A middleware infrastructure to enable active spaces.IEEE Pervasive 1:74-82

[18] Roman M, Kon F and Campbell R H (2001) Reflective middleware: From your desktop to your hand.IEEE Distributed Systems Online. Special Issue on Reflective Middleware

[19] Smith B C (1984) Reflection and semantics in lisp. Proc. 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Salt Lake City, Utah, United States,23-25

[20] Sousa J P and Garlan D (2002) Aura: An architectural framework for user mobility in ubiquitous computing environments. Proc. IEEE/IFIP Conference on Software Architecture, Montreal,29-43

[21] Tandler P (2001) Software infrastructure for ubiquitous computing environments: Supporting synchronous collaboration with heterogeneous devices. Proc. Ubicomp 2001: Ubiquitous Computing, Atlanta, Georgia,96-115

[22] Weiser M (1991) The computer for the twenty-first century.Scientific American 265:94-101