# A Middleware Infrastructure for Active Spaces

*The Gaia metaoperating system extends the reach of traditional operating systems to manage ubiquitous computing habitats and living spaces as integrated programmable environments.*

**Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt**
*University of Illinois at Urbana-Champaign*

Pervasive computing environments augment human thought and activity with digital information, processing, and analysis, providing an observed world that is enhanced by the behavioral context of its users. A spectrum of heterogeneous computation and communication devices aim to enhance user productivity and facilitate everyday tasks. Despite the prevalence of such devices, however, no suitable software infrastructure with which to develop applications for ubiquitous computing habitats or living spaces exists.

To address this deficiency, we developed Gaia, a metaoperating system[1] (detailed in the sidebar "The Gaia Operating System") built as a distributed middleware infrastructure that coordinates software entities and heterogeneous networked devices contained in a physical space. Gaia is designed to support the development and execution of portable applications for *active spaces*[2]—programmable ubiquitous computing environments in which users interact with several devices and services simultaneously. Gaia exports services to query, access, and use existing resources and context, and provides a framework to develop user-centric, resource-aware, multidevice, context-sensitive, and mobile applications.

By extending the concepts of traditional operating systems to ubiquitous computing spaces, we can simplify space management and application development. Gaia's main contribution is not its individual services but rather the functionality it provides as the result of the interaction of these services. This interaction lets users and developers abstract ubiquitous computing environments as a single reactive and programmable entity instead of a collection of heterogeneous individual devices. In this article, we present an overview of the Gaia architecture, focusing on the complete system rather than individual services.

## Active spaces

A physical space, illustrated in Figure 1a, is a geographic region with limited and well-defined physical boundaries containing physical objects, heterogeneous networked devices, and users performing a range of activities. An active space, shown in Figure 1b, is a physical space coordinated by a responsive context-based software infrastructure that enhances mobile users' ability to interact with and configure their physical and digital environments seamlessly. Active spaces must support the development and execution of user-centric mobile applications.

In active spaces, *sessions* associate user data and applications with users. This lets users move across active spaces and have their data and applications always available. When a user enters an active space, the user's sessions are dynamically mapped to the active space resources. Users can define different sessions and can activate and suspend them as necessary. We refer to the collection of sessions associated with a user as the *user virtual space*. The user virtual

1536-1268/02/$17.00 © 2002 IEEE

## The Gaia Operating System

The motivation behind the Gaia operating system is that of traditional operating systems, but at another level of abstraction. The Gaia OS abstracts a space and all the resources it contains as a single programmable entity. Abraham Silberschatz and his colleagues define seven services common to all operating systems: program execution, I/O operations, file-system manipulation, communications, error detection, resource allocation, and accounting and protection.[1] The Gaia OS currently provides the first six services, and we are completing a security prototype.

### Program execution

The Gaia OS *component management core* lets applications create, destroy, and upload components in any execution node in the active space. CMC uses the program execution facilities of the execution node's OS, which includes memory, thread, and process management.

### I/O operations

Gaia's OS leverages the low-level OS I/O functionality and provides device drivers (implemented as distributed objects) to export it to the rest of the active space. Gaia's OS also defines default I/O channels (input, output, and error) and maps them to event channels, allowing the creation of a default space "console."

### File-system manipulation

The *context file system* lets users, services, and applications manipulate files in active spaces. CFS interacts with devices' low-level OS file systems to access and export data to the active space. File locations are hidden from users, but users can access them from any device in the active space. CFS extends traditional OSs with functionality to dynamically transform data to different formats and to organize data by context.

### Communications

The Gaia OS supports direct communication, which is similar to synchronous low-level OS interprocess communication (IPC), and indirect communication, which is similar to asynchronous low-level OS IPC. The Gaia OS supports remote procedure calls (RPC) for direct communication and events (suppliers and consumers) for indirect communication. In both cases, Gaia's OS leverages standard communication middleware.

Events, which are similar to Unix signals, notify entities in the active space about added or removed resources, error conditions, file system changes, and application state changes. The use of asynchronous events improves system reliability by decoupling event producers from event listeners. Gaia entities use other mechanisms, such as RPC for remote method invocation and nonblocking streaming for audio and video transmission.

### Error detection

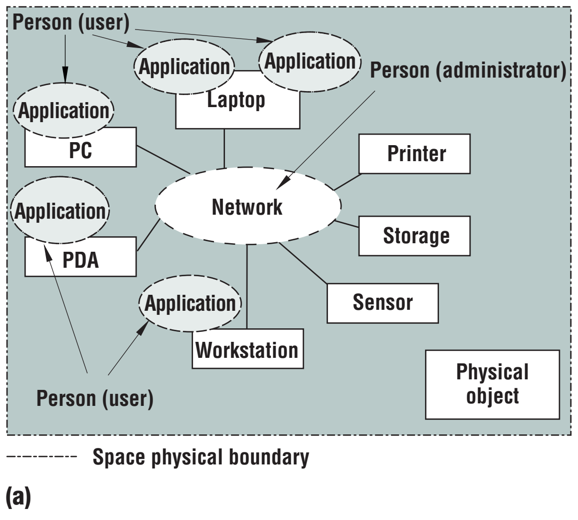
Error detection includes both software and hardware errors that affect application execution. The Gaia OS uses the event service to report errors. Users register services that receive error notifications and react accordingly (for example, they might notify users, restart components, or suspend applications).

### Resource allocation

In a traditional OS, resource allocation relates to managing hardware resources such as memory, CPU, and disk. The Gaia OS extends the notion of resource allocation to devices, services, and applications in the active space.

### REFERENCE

1. A. Silberschatz and P. Galvin, *Operating System Concepts*, 5th ed., Addison-Wesley, Boston, 1998.



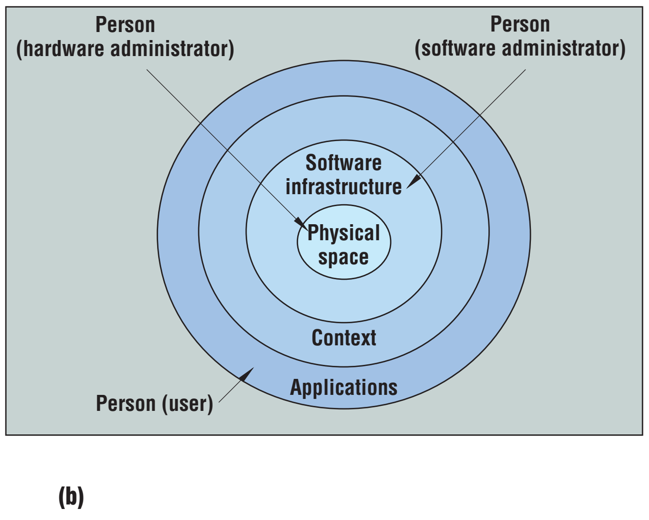
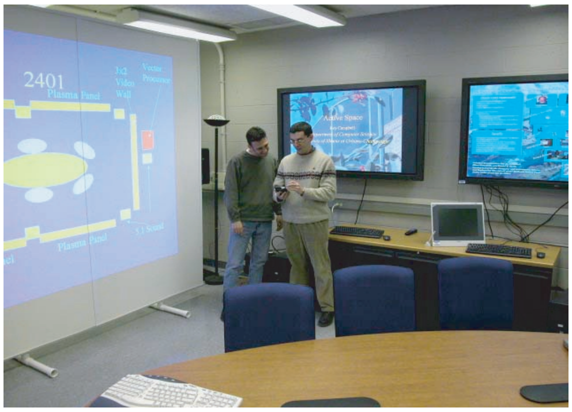


Figure 1. Physical and active spaces. (a) A physical space contains physical objects, networked devices, and users within well-defined physical boundaries. (b) An active space extends the physical space, adding coordination via a context-based software infrastructure.



(a)

| | |
|---|---|
| 15 | Pentium IVs at 1.2GHz |
| 4 | NEC HD plasma displays (61") |
| 1 | Sharp HD projector |
| 2 | Sound Web sound system |
| 4 | Infrared beacons |
| 2 | iButton detectors |
| 5 | Touch-screen displays |
| 4 | Badge detectors |
| 3 | Compaq iPaq handheld devices |
| 1 | Gbit Ethernet network |
| 1 | Wavelan wireless network at 11 Mbps |
| | X10 appliance controllers |

(b)

Figure 2. (a) Our prototype Gaia-enabled active space and (b) a list of the room's equipment. The active meeting room serves as a testbed for our software infrastructure during meetings and seminars.

space requires support to locate resources available in the user's environment and to map the sessions to the existing resources.

Active spaces such as the one depicted in Figure 2 challenge existing assumptions for traditional PC applications. As Marc Weiser observed, ubiquitous computing requires systems that vanish into the background.[3] In an active space, a one-to-one relationship between a user and keyboard, mouse, and display interfaces does not exist. Indeed, the complexity of ubiquitous applications encourages a relationship between a user and an active space. Active-space system-software support should simplify application programming and execution. Much like PC applications rely on operating systems, active-space applications need an OS that supports access to and operation of the resources contained in the space that hosts their execution. The sidebar "Related Work in Active Environments" describes other work in this area.

## System software

Gaia manages an active space's resources and services; provides location, context, and event services; and stores information about the active space. Gaia has three major components, as Figure 3 shows: the kernel, the application framework, and the applications.

### Gaia kernel

The Gaia kernel consists of a *component management core* and an interrelated set of basic services used by all Gaia applications. The CMC dynamically loads, unloads, transfers, creates, and destroys all Gaia components and applications. Because they are component-based, distributed, and mobile, Gaia applications require support for remote component execution and management. Remote execution nodes register with the active space and host the execution of Gaia components.

The current implementation of Gaia uses Corba; however, the system could also use SOAP, RMI, or customized communication middleware. Although Corba provides a stable infrastructure for distributed object interaction, active spaces require extensions to handle soft state, dynamic resource detection, and fault tolerance. Gaia offers five basic services:

- Event manager
- Context service
- Presence service
- Space repository
- Context File System

Some of the services, such as the event manager, are built on top of existing middleware services, while others, such as the presence service, extend the communication middleware.

***Event manager.*** Active spaces require a flexible mechanism to expose changes in their current state—for example, when a component starts, an application moves, or a user enters the active space. The event manager service distributes events in the active space and implements a decoupled communication model based on suppliers, consumers, and channels. Event channels forward suppliers' events to the consumers registered with the channel. The event manager has a single entry point and one or more event channel factories. Each event factory creates channels for specific types of events—for example, high-speed or persistent events. A default set of event channels notifies interested Gaia components about new services, applications, people, errors, and component heartbeats. Applications can also define their own event channels for application state changes. The event service lets applications tap into event channels to learn about changes in the system.

By decoupling information suppliers from information consumers, the event manager increases system reliability. If a supplier fails, the service can automatically replace it with a replica that continues delivering messages to its assigned channel without disrupting the system. Our current event manager implementation uses Corba's event service[4] as the default event factory. A more detailed description of the event manager service is available elsewhere.[5]

***Context service.*** Gaia applications can use context information to adapt to user behaviors and activities.[6] Our context service lets applications query and register for particular context information, which helps them adapt to their environment. The context infrastructure consists of several components, or *context providers*, that offer infor-

## Related Work in Active Environments

The Microsoft EasyLiving project focuses on home and work environments.[1] The main properties of these environments are self-awareness, casual access, and extensibility. The project infrastructure lets interfaces move with the user. Easy Living uses computer vision to recognize gestures and users and to detect user location. The system uses this information to customize the room. Whereas Easy Living sees the desktop as the primary user interface, Gaia allows application partitioning on different devices.

The i-Land[2] and Roomware[3] research projects present infrastructures that digitally augment meeting rooms. Both projects aim to make it easy for users to exchange ideas, digitally record meeting results, and search knowledge bases, and both provide multimedia data exchange tools for group collaboration. Stanford's Interactive Workspaces presents an augmented meeting room that promotes group work.[4] The room contains wall-sized touch screens, projectors, microphones, speakers, laptops, and PDAs. Coordinating the entities in the room requires a high-level operating system.

Roomware, i-Land, and Interactive Workspaces look at users' interactions with physical spaces (mostly meeting rooms) and collaborative work groups. Like the developers of Roomware and Interactive Workspaces, we believe that a need for a supporting infrastructure exists. However, we focus on generic spaces (office and house, for example), which might or might not imply collaborative work. While some active spaces define a collaborative environment, others are mostly single-user based. Furthermore, Gaia expects mobile users to move their applications and data across different active spaces.

Aura shares several design goals with Gaia.[5] It emphasizes the notion of mobile users moving among environments and defines an environment similar to our active spaces. Tasks identify user-associated applications that can migrate from one environment to another. A software infrastructure supports task execution, maximizing available resources and minimizing user distraction. The main difference between Gaia and Aura is that Gaia emphasizes space programmability. Gaia allows users to configure their applications to benefit from the resources in their current space. Users can interact with multiple devices simultaneously, reconfigure applications dynamically, suspend and resume groups of applications, and program application behavior based on context attributes.

### REFERENCES

1. B. Brumitt et al., "EasyLiving: Technologies for Intelligent Environments," *Proc. Handheld and Ubiquitous Computing* (HUC), Springer-Verlag, Heidelberg, 2000, pp. 12–29.
2. P. Tandler, "Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices," *Proc. Ubiquitous Computing* (Ubicomp 2001), Springer-Verlag, Heidelberg, 2001, pp. 96–115.
3. N. Streitz, J. Geissler, and T. Holmer, "Roomware for Cooperative Buildings: Integrated Design of Architectural Spaces and Information Spaces," *Proc. Workshop on Cooperative Buildings* (CoBuild 1998), Springer-Verlag, Heidelberg, 1998, pp. 4–15.
4. A. Fox et al., "Integrating Information Appliances into an Interactive Workspace," *IEEE Computer Graphics & Applications*, vol. 20, no. 3, May 2000, pp. 54–65.
5. J.P. Sousa and D. Garlan, "Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments," *Proc. IEEE Conf. Software Architecture*, IEEE CS Press, Los Alamitos, Calif., 2002, pp. 29–43.

mation about the current context. These include sensors that track people's locations, room conditions (for example, temperature and sound), weather, and current stock prices. Other components can infer certain higher-level contexts on the basis of sensed information. For example, one component deduces the type of activity occurring in a room (a meeting, presentation, or movie screening, for example) on the basis of who is in the room, which applications are running, and other cues. A registry maintains a list of available context providers. Applications can use this registry to find providers of the contexts they desire.

We base our context model on first-order logic and Boolean algebra, which lets us easily write rules to describe context information. We represent context using a first-order predicate with four arguments. We borrow this structure of a context predicate from a simple clause in the English language of the

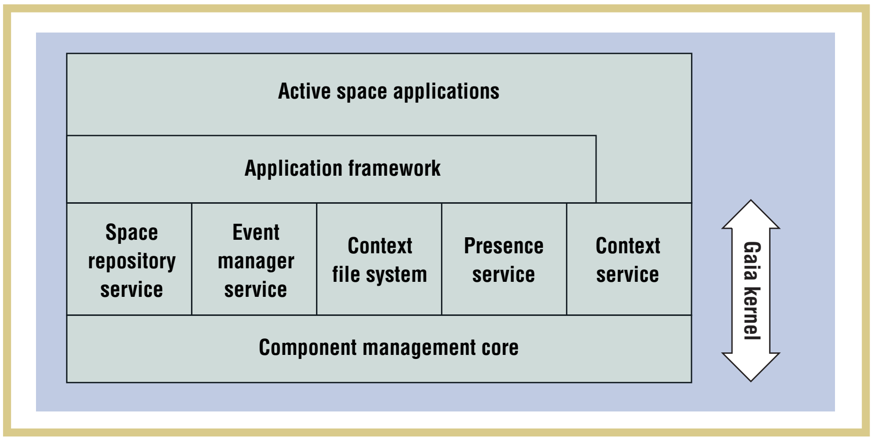


Figure 3. Gaia architecture. The architecture consists of a kernel, which includes a component management core and a set of basic services, a component-based application framework, and active space applications.

form <subject><verb><object>. An atomic context predicate is defined as Context(<ContextType>, <Subject>, <Relater>, <Object>). The *context type* refers to the context the predicate is describing; the *subject* is the person, place, or thing with which the context is concerned; and the *object* is a value associated with the subject. In our implementation, ContextType maps to an event channel. The *relater* associates the subject and the object much like a comparison operator (=, >, or <), verb, or preposition. Example context predicates are Context(temperature, room 3231, is, 98 F) and Context(printer status, srgalw1 printer queue, is, empty). In some cases, one or more predicate elements might be empty.

When a digital entity fails to send the heartbeat, the digital entity presence subsystem assumes that it is no longer available and notifies the rest of the space that the entity left.

We can construct more complex contexts by performing first-order logic operations such as quantification, implication, conjunction, disjunction, and context predicate negation. One example of a rule is Context(number of people, room 2401, >, 4) AND Context(application, Powerpoint, is, running) => Context(social activity, room 2401, is, presentation).

Anind K. Dey's and Gregory Abowd's Context Toolkit inspired our context infrastructure.[7] We structure the expressive power of contexts with first-order logic to frame rules and queries and to infer properties involving context using mechanisms similar to those of Prolog and other automated theorem provers. We can determine high-level context information from low-level context information, much like the Context Toolkit's aggregators. Our system also formalizes the exchange of context information among system components and lets us describe component properties.

***Presence service.*** As a resource-aware infrastructure, Gaia must maintain updated information about active space resources. Our presence service differs from existing context infrastructures (such as the Context Toolkit) in that it detects and maintains soft-state information about software components, devices, and people. The service is divided into two main subsystems: digital-entity presence and physical-entity presence.

Gaia currently defines four basic types of entities: application, service, device, and person. The digital entity presence subsystem detects service and application entities, which periodically send heartbeats to notify the service that they are in the active space. When a digital entity fails to send the heartbeat, the digital-entity presence subsystem assumes that it is no longer available—either it was stopped or it crashed—and notifies the rest of the space that the entity left.

The physical-entity presence subsystem detects devices and people present in the active space. This subsystem uses different types of sensors to proactively detect physical entities and, if possible, their locations. The physical-entity presence subsystem implements a beaconing mechanism on behalf of the physical entities, acting as a proxy. It is implemented as an open infrastructure in which we can incorporate different sensor device drivers and algorithms.

***Space repository.*** Active space entities need to know the resources present in the active space and their properties. The space repository stores information (name, type, owner, and so on) about all software and hardware entities in the space and lets applications browse and retrieve entities on the basis of specific attributes. The space repository learns about entities entering and leaving the active space through the presence service channels.

Applications use the space repository during instantiation to find suitable resources. When an application starts executing, for example, it uses the space repository to find appropriate resources (such as execution nodes, displays, and speakers). This level of indirection lets us describe applications generically (active-space independent) and map them to resources in different active spaces.

All active-space resources are associated with an XML description that contains their properties. When a new resource enters the active space, the space repository contacts it to obtain its XML description. The current version of the space repository uses a Corba Trader[4] to store entity data. We currently use the Trader's constraint query language, but we plan to extend it with a generic language that could be mapped either to the Corba Trader constraint language or to standard SQL. For example, the query Category = = 'Device' and Type = = 'Display' returns a list of all displays in the active space.

***Context File System.*** Active spaces are often designated for specific tasks. Applications can use task context to distinguish meaningful from irrelevant information. Long-running processes might not have the luxury of human intervention to locate required data, which can vary over time owing to context changes. In addition, users are highly mobile in active spaces and should not have to manually transfer files or data between environments. The environment should assist users in making personal storage automatically available in the users' current location.

To address these issues, we have developed *Context File System*, a context-aware file system that uses application-defined properties and environmental context information to simplify many of the tasks that are traditionally performed manually or require additional programming. More specifically, the CFS uses context to

- Make personal data automatically available to applications, conditioned on user presence
- Organize data to facilitate locating relevant material for applications and users
- Retrieve data in a format based on user preferences or device characteristics through dynamic data types

The CFS constructs a virtual directory hierarchy,[8] based on the types of context associated with particular files, and aggregates related material. We implement the directory hierarchy layout using a mounting mechanism in which users own mount points, which contain context tags. Users can merge their personal mount points into a space to make their data available to applications and other users. The CFS is aware of different types of context, which the context service, users, and applications define.

The CFS presents context as directories, in which path components represent context types and values. The file system path syntax uses the context service's quaternary predicate structure, in which the relater defaults to equality. We can attach context to files and directories by copying data to a context directory, which associates the context with the data. The virtual directory hierarchy forms a simple query language to determine what types of context are attached to files. For example, to determine which files have the associated context location == RM2401 && situation == meeting, we enter /location:/RM2401/situation:/meeting directory. The system is a hybrid database and file system; the database functionality offers the flexibility to search for relevant information, and the file system functionality provides a familiar interface for application developers.

The CFS combines the environment's current context properties (location, time, situation, and so on) with user-specified properties to display the correct application data. For example, a seminar application might require all papers to be discussed during a seminar. A calendar or the moderator's arrival can trigger the application; thus, the application must be able to locate the correct files to display. The application simply opens the directory for the current papers (say, /type:/papers/current). The file system uses the current location, situation, and time along with the user-specified request, "papers," to find the correct files for the application. The directory's contents might change automatically every week, as new papers are added and others time out. From the application's viewpoint, however, it simply opens the same directory every week and finds the relevant material.

The space might also define a context that is irrelevant to the current task. The context "the weather is sunny," for example, might be meaningless to the seminar application, but might make sense for a telepresence application. The system resolves this issue by ignoring any context that is valid in the environment but not explicitly associated with the data. Although we view the context directory structure as a hierarchy, context directories impose no fixed ordering. Context paths can be traversed in any order.

The CFS architecture is composed of mount and file servers. One mount server maintains an active space's namespace. The namespace changes as users physically move in and out of the space. File servers export storage to the local space from local or remote locations. We implement our servers at the application-level and leverage existing native file systems to access files and directories. You can find more details about the CFS elsewhere.[9]

## Application framework

In a Gaia active space, applications are partitioned among a group of coordinated devices.[10] Applications receive input events from devices, services, users, and other applications; present their state using different types of devices (for example, sound system, display, and temperature-control devices); and adapt to changes in the environment. Active spaces let users decide how to interact with applications using a number of inputs, outputs, and processing devices.

Developing applications for active spaces is challenging. For example, an application might need to move the output data from one display to another, duplicate the output to different displays, transform a visual representation into speech, and switch from a mouse to a voice input sensor (a hardware or software entity that triggers changes in the application), all of which it must perform while providing application-consistency guarantees. Our application framework lets users construct, run, or adapt existing applications to active spaces. The framework consists of

- A distributed component-based infrastructure, which reuses the application partitioning proposed by the traditional Model-View-Controller (MVC)[11] and introduces new functionality to export and manipulate the application component bindings.
- A mapping mechanism, which customizes applications to active spaces.
- A group of policies that defines sets of rules for customizing several aspects of applications, including instantiation, mobility, reliability, and composition (number of components and their bindings).

***Infrastructure.*** The application infrastructure has four components. The first three are the building blocks for any application:

- The *model* implements the application logic.
- The *presentation* exports the application data.
- The *input sensor* lets the user interact with the application.

Active spaces are often designated for specific tasks. Applications can use task context to distinguish meaningful from irrelevant information.

- The *controller* maps input sensor events (such as touch-screen events and context changes) into method requests for the model.
- The *coordinator* manages the first four components.

The model, presentation, and controller are strictly related to application-domain functionality, while the coordinator pro-

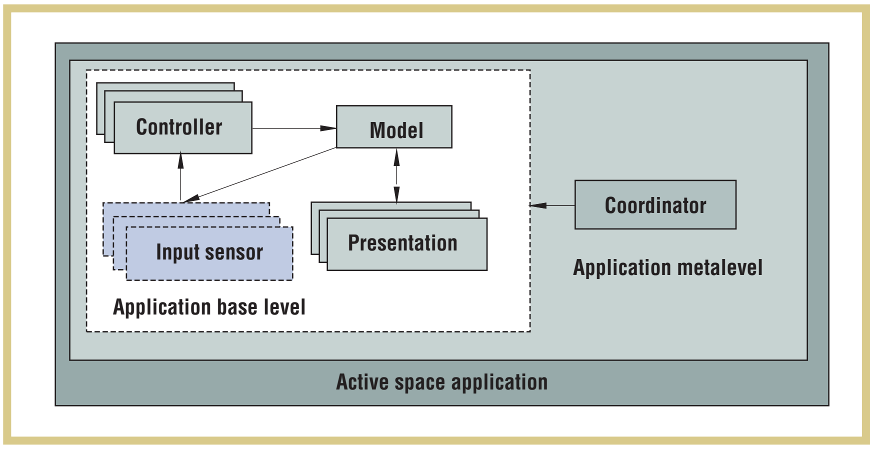


Figure 4. Application framework infrastructure. The coordinator oversees the composition of the model, presentation, and controller components.

vides the application's metalevel functionality. Figure 4 illustrates the framework infrastructure.

***Mapping mechanism.*** Active-space heterogeneity requires a mechanism for customizing applications to different scenarios. For example, a calendar application running in an active office might simultaneously use a plasma display to show the week's appointments, a handheld to display the day's appointments, and an input sensor running on a PC to enter data. The same calendar running in an active car might use the car's sound system to broadcast information about the next appointment, and a speech-recognition-based input sensor to query the calendar and to enter and delete data.

The mapping mechanism defines two application description files: an *application generic description* and an *application customized description*. An AGD, which is created by the application developer, is an active space-independent description listing the application components, the minimum and maximum number of instances allowed, and component requirements (for example, audio output and Windows OS). An ACD contains a list of application components, including their associated execution nodes (chosen according to component requirements) and initialization parameters. An ACD is implemented as a script that coordinates the instantiation and assembly of the different components. A specialization mechanism uses an AGD and the target active space's space repository service to generate ACDs. The mapping mechanism uses the space repository to find devices and services that meet AGD requirements.

***Customization policies.*** The application framework infrastructure and the mapping mechanism provide the tools to build and instantiate applications. The application framework relies on policies to address reliability, adaptation, mobility, and related issues. Users can define their own policies or can use the framework's default policies.

***MVC extensions.*** In addition to adding a coordinator component to manage application component composition, we have introduced three changes to MVC to accommodate our requirements for environmental awareness, application partitioning, context sensitivity, user-centrism, and mobility:

- We define a new component, *presentation*, that models any output representation, not only graphical representations as the MVC view proposes.
- We generalize the definition of the MVC input sensor (a hardware device) to incorporate software components (a context input sensor, for example).
- We generalize the input sensor time-sharing model defined by the MVC into a space–time-sharing model.

In MVC, all application views and controllers share input sensors (a mouse and a keyboard, for example), and thus the OS must schedule the input sensors. Graspable interfaces[12] enable space sharing, which avoids input sensor scheduling by assigning different input sensors to the application's different functions. We combine the two approaches into space–time sharing to model active space applications. A music application running in an active space might use a PDA to control the current song and speech recognition to control the volume (space sharing); however, the same space might host a calendar application that uses the PDA to browse appointments and uses speech recognition to control the calendar's functionality (time sharing). Space–time sharing allows more than one active controller and presentation. This is unlike MVC, in which only one controller–view pair can be active at any time. More details about the application framework are available elsewhere.[13]

## Gaia management tools

Gaia relies on a scripting language to coordinate the digital entities running in an active space. This language simplifies overall active-space management. For example, we use a script to implement the bootstrap mechanism that starts the Gaia OS execution in a physical space.

### Scripting language

Gaia uses a high-level scripting language, LuaOrb, to program and configure active spaces and to coordinate the active entities they contain.[14] LuaOrb is based on the interpreted language Lua, which simplifies management and configuration tasks and allows for rapid prototyping and testing.[15] The Lua interpreter is fast and has a small memory footprint, which makes it suitable for resource-constrained devices. LuaOrb implements language bindings between Lua and Corba, COM, and Java. LuaOrb's ability to communicate with various component models lets it easily interact with our system compo-

Figure 5. An example Lua script for instantiating and assembling an MP3 application.

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

nents. We use Lua to implement the bootstrap algorithm, instantiate applications, interact with execution nodes to create components and easily glue them together, and quickly test components and applications.

The example script in Figure 5 instantiates and assembles an MP3 application. The script uses the Gaia space repository to obtain a handle to an audio output device (line 1); an execution node for the model (line 2); an execution node for the coordinator (line 3); and a touch screen, "plasma 1," for the input sensor (line 4). It then uses the CMC functionality to create the coordinator (line 5), model (line 6), presentation (line 7), and input sensor (line 8). Finally it assigns the model to the coordinator (line 9) and registers the presentation (line 10) and the input sensor (line 11) with the application, using the interface exported by the coordinator.

Although we achieve the same result with C++ and Java, they require more code, time, and user effort. Lua effectively simplifies the manipulation and coordination of entities.

### Bootstrap protocol

Gaia implements a bootstrap protocol that interprets a configuration file (Lua script) and starts the kernel services accordingly. The configuration file contains information about the Gaia kernel services, such as the service name, the name of the service interface, the Gaia node or nodes that will host the service, the service instantiation policy, and start parameters. Individual Gaia kernel services can also specify additional configuration parameters. Currently, the active-space administrator provides the list of devices in the space. In the future, we expect automatic device discovery via different sensor technologies.

Figure 6 illustrates a state transition diagram with the instantiation order of the Gaia kernel services. The configuration file lists primary and backup Gaia nodes for each Gaia kernel service; the bootstrap process uses the dependencies diagram and the configuration file to decide whether or not to start a service in a particular Gaia node. Gaia uses a time-out mechanism and a probing protocol to ensure that each Gaia service is started in at most one machine (as specified in the configuration file).

## Using Gaia

We developed the presentation manager application to demonstrate Gaia's functionality. The application lets users present slides in multiple displays simultaneously, move and duplicate slides to different displays during the presentation, and move and duplicate the input sensor controlling the presentation to different devices.[16] The presentation manager uses Microsoft's PowerPoint to manipulate slides (using the COM interface).

Our experience with the application shows that most users edit the presentation in their offices and use the CFS to import the data into the active meeting room. The most common interaction mechanism is a wireless-enabled handheld device (using Bluetooth) running a software input sensor with buttons for start, stop, next, and previous.

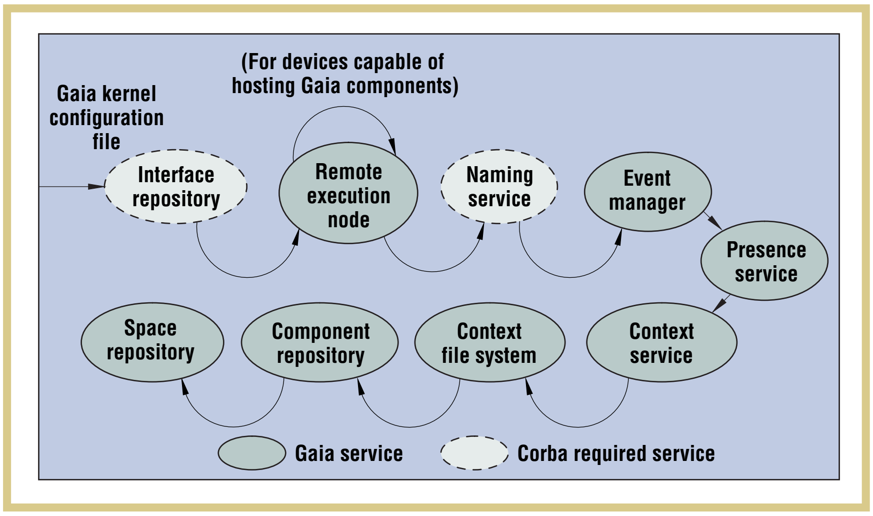


Figure 6. Gaia kernel bootstrap sequence. Solid green circles denote Gaia kernel services, and dotted yellow circles denote middleware-specific services.

## Registering with the active space

When a speaker enters a Gaia-enabled room carrying a handheld device and an RF active badge, the presence service detects the badge and sends an event to the person-presence channel. This event contains information about the speaker, including a reference to his or her profile. The space repository receives the event, retrieves an XML description for the user entity, and stores the information. The CFS also receives the event about the new user, accesses the user profile, obtains the speaker's mount points, and mounts the data in the space. The slide show file stored in the speaker's active office is now accessible from the active meeting room (see Figure 2).

Next, the speaker registers the handheld with the space, so that he or she can use it to control the presentation. The room is equipped with infrared beacons that broadcast the space name and a handle to a directory service. This directory service (a Corba naming context) contains references to the Gaia kernel services. The handheld device picks up the infrared beacon, resolves the event manager from the directory, and initiates the beaconing mechanism, which periodically sends a heartbeat event to the device heartbeat channel. The presence service receives the event and sends a new event to the device presence channel to notify the rest of the space about the new device. The space repository receives the event, contacts the device to retrieve the XML description, and stores the information. Both the speaker and the handheld device are now entities of the active space. They are stored in the space repository; other entities can contact them; and they can use the space resources.

## Starting the application

The active meeting room runs an application that triggers actions according to user-specified conditions. We configure this application to automatically start the presentation manager application when the speaker enters the room. The application registers with the context service to be notified when the room context meets this condition. The trigger service entry is a Lua script that gets the presentation file's name from the `/type:/presentation/current:` context directory and starts the presentation manager. The context associated with the slide show presentation file includes the entry `location=2401`. Therefore, when the user data is mounted in Active Meeting Room 2401, the file is visible from `/type:/ppt/current:`. The Lua script stored in the trigger service requires a valid ACD to start the application. The script uses the file system and accesses `/type:/lua/acd:/gpm/current:`, which contains presentation manager ACDs customized for 2401. The Lua script chooses the ACD "default."

The ACD is also a Lua script that interacts with the CMC to instantiate the components. The ACD contacts the application coordinator and registers the model, the presentations, the controllers, and the input sensors. The model interacts with the event manager to create a channel that it uses to send update messages to the presentations and registers the presentations with the channel. The coordinator assigns the model's reference to the presentations, and the controller's reference to the input sensors. All the application components initiate the beaconing mechanism and are therefore detected by the presence service, introduced to the space using an "entered" event, and registered in the space repository.

## Interacting with the application

The default presentation manager ACD creates the application input sensor in one of the room's touch screens. Say the speaker moves the input sensor to his or her handheld device. To move application components, we provide a library that interacts with the space repository to locate the handheld and create a new instance of the input sensor using the CMC. Next, the library locates the application coordinator in the space repository, registers the new input sensor, unregisters the original sensor, and then uses the CMC to delete the original input sensor. The presence service stops receiving heartbeats from the original input sensor and sends an event to the service-presence channel to notify it that the entity has left. The space repository removes the information about the entity. Using the library, the speaker can move the presentation slides to new displays in the middle of a presentation. New displays might be the PDAs of people entering the meeting room.

The presentation manager uses four plasma displays simultaneously. When the user selects "start" on the handheld's input sensor, the input sensor sends a request to the application controller, which sends a `startPresentation` request to the model. The model sends a start event to the application updates channel. As a result, all presentations contact the model to obtain a handle to the presentation file. Leveraging the CFS, the model opens the file and returns a handle. The presentations get the file and display the first presentation slide. When the users selects "next," the input sensor sends a request to the controller, which sends a `nextSlide` request to the model. The model sends a next event to the application channel, which instructs presentations to display the next slide.

We plan to develop new applications to validate different aspects of Gaia. We will also extend the infrastructure with a security service and expand our current service implementation to support the notion of a virtual-space abstraction. The virtual space lets users create active-space templates that the Gaia OS can dynamically map to real spaces. For example, a virtual active meeting room contains a projector, auxiliary displays, and a touch screen, and has an associated context model with properties relevant to a meeting. Users can assign applications to the virtual spaces, which are mapped to virtual devices in the room. The virtual-space mapping mechanism interacts with an active space to find appropriate resources on the basis of the virtual space requirements and moves the applications from the virtual space to the active space. Finally, we are also studying how to federate Gaia services to aggregate different active spaces.

## ACKNOWLEDGMENTS

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

## REFERENCES

1. F. Kon et al., "2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments," *Proc. ECOOP 98 Workshop Reflective Object-Oriented Programming and Systems*, Springer-Verlag, Hidelberg, Germany, 1998, pp. 388–390.

2. M. Román and R.H. Campbell, "GAIA: Enabling Active Spaces," *Proc. 9th SIGOPS European Workshop*, ACM Press, New York, 2000, pp. 229–234.

3. M. Weiser, "The Computer for the Twenty-First Century," *Scientific Am.*, 1991, pp. 94–101.

4. M. Henning and S. Vinosky, *Advanced Corba Programming with C++*, Addison-Wesley, Boston, 1999.

5. B. Borthakur, *Distributed and Persistent Event System for Active Spaces*, master's thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 2002.

6. B.N. Schilit, N. Adams, and R. Want, "Context-Aware Computing Applications," *Proc. IEEE Workshop Mobile Computing Systems and Applications*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 85–90.

7. A.K. Dey and G.D. Abowd, "The Context Toolkit: Aiding the Development of Context-Aware Applications," *Proc. Workshop Software Eng. for Wearable and Pervasive Computing*, ACM Press, New York, 2000, pp. 434–441.

8. D.K. Gifford et al., "Semantic File Systems," *Proc. 13th ACM Symp. Operating Systems Principles* (SOSP13), ACM Press, New York, 1991, pp. 16–25.

9. C.K. Hess, *A Context File System for Ubiquitous Computing Environments*, tech. report UIUCDCS-R-2002-2285 UILU-ENG-2002-1729, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, July 2002.

10. B.A. Myers, "Using Hand-Held Devices and PCs Together," *Comm. ACM*, vol. 44, no. 11, Nov. 2001, pp. 34–41.

11. G.E. Krasner and S.T. Pope, *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*, tech. report, ParcPlace Systems, Mountain View, Calif., 1988.

12. G.W. Fitzmaurice, *Graspable User Interfaces*, doctoral thesis, Computer Science Dept., Univ. of Toronto, 1996.

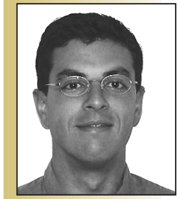
13. M. Román and R.H. Campbell, *A User-Centric, Resource-Aware, Context-Sensitive, Multi-Device Application Framework for Ubiquitous Computing Environments*, tech. report UIUCDCS-R-2002-2284 UILU-ENG-2002-1728, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, July 2002.

14. R. Cerqueira, C. Cassino, and R. Ierusalimschy, "Dynamic Component Gluing across Different Componentware Systems," *Proc. Int'l Symp. Distributed Objects and Applications* (DOA 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 362–373.

15. R. Ierusalimschy, L. Figuereido, and W. Celes, "Lua: An Extensible Extension Language," *Software: Practice and Experience J.*, vol. 26, no. 6, 1996, pp. 635–652.

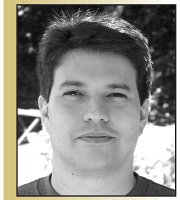
16. C.K. Hess, M. Román, and R.H. Campbell, "Building Applications for Ubiquitous Computing Environments," *Proc. Int'l Conf. Pervasive Computing* (Pervasive 2002), Springer-Verlag, Heidelberg, Germany, 2002, pp. 16–29.

## the AUTHORS

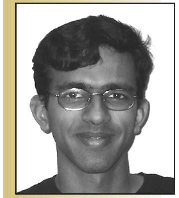


**Manuel Román** is a PhD candidate at the University of Illinois at Urbana-Champaign. His research interests include ubiquitous computing, middleware, operating systems, and interactive and programmable active spaces. He received his BS and MS in computer science from the La Salle School of Engineering (Ramon Llull Univ.). Contact him at 3134 DCL, 1304 W. Springfield Ave., Urbana, IL 61801; mroman1@cs.uiuc.edu.

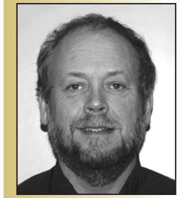
**Christopher Hess** is a PhD candidate in the Computer Science Department at the University of Illinois at Urbana-Champaign. His interests include ubiquitous computing systems, streaming media, networking, operating systems, and object-oriented design. Hess received his BS and MS in mechanical engineering from Tufts University and an MS in computer science from the University of Illinois. Contact him at 3134 DCL, 1304 W. Springfield Ave., Urbana, IL 61801; ckhess@cs.uiuc.edu.



**Renato Cerqueira** is an assistant professor of computer science at the Pontifical Catholic University of Rio de Janeiro. He is also a research scientist and project manager at the Computer Graphics Technology Group of PUC-Rio (Tecgraf/PUC-Rio). His research interests include component-based development, object-oriented languages, middleware platforms, distributed programming, software architectures, and distributed simulation. He received his MSc and PhD in computer science from the Pontifical Catholic University of Rio de Janeiro. He is a member of the ACM and the IEEE Computer Society. Contact him at the Computer Science Dept./PUC-Rio, Rua Marques de Sao Vicente 225, 407-RDC, Rio de Janeiro - RJ, Brazil 22453-900; rcerq@inf.puc-rio.br.



**Anand Ranganathan** is pursuing a PhD in computer science at the University of Illinois at Urbana-Champaign. His research interests include context gathering and modeling, using contextual information to make applications more intelligent in their interactions with humans, security and data management in pervasive computing environments, mobile computing, and wearable computing. He received a BTech in computer science from the Indian Institute of Technology in Madras, India. Contact him at 3134 DCL, 1304 W. Springfield Ave., Urbana, IL 61801; ranganat@cs.uiuc.edu.



**Roy H. Campbell** is a professor of computer science at the University of Illinois at Urbana-Champaign. His research interests include operating systems, distributed multimedia, network security, and ubiquitous computing. He received his BSc in mathematics from the University of Sussex, and his MSc and PhD in computing from the University of Newcastle upon Tyne. Contact him at 3134 DCL, 1304 W. Springfield Ave., Urbana, IL 61801; rhc@cs.uiuc.edu.

**Klara Nahrstedt** is an associate professor and Fisher Professor at the University of Illinois at Urbana-Champaign. Her research interests include multimedia middleware, protocols, operating system support for multimedia, software engineering for distributed multimedia applications, and availability and quality of service on high-speed and mobile networks, as well as on high-end and handheld computers. She received a PhD from the University of Pennsylvania. Contact her at 3134 DCL, 1304 W. Springfield Ave., Urbana, IL 61801; klara@cs.uiuc.edu.