© Copyright by Christopher K. Hess, 2003
THE DESIGN AND IMPLEMENTATION OF A CONTEXT-AWARE FILE SYSTEM
FOR UBQUITOUS COMPUTING APPLICATIONS

BY

CHRISTOPHER K. HESS
B.S., Tufts University, 1991
M.S., Tufts University, 1993
M.S., University of Illinois at Urbana-Champaign, 1998

THESIS

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

Urbana, Illinois
Abstract

Ubiquitous computing environments stretch the requirements of traditional infrastructures used to facilitate the development of applications. Activities are often supported by collections of applications, some of which are automatically launched with little or no human intervention. This task-driven environment challenges existing application construction and data management techniques. One of the distinguishing features of ubiquitous computing is the integration of context into the system. Context, such as time, location, and situation, allows applications to adapt to the current surroundings in order to facilitate the use of the computational environment. In this thesis, we present a file system for ubiquitous computing applications that is context-aware. Context may be associated to files and directories and is used to limit the scope of available data to what is important for the current task, aggregates related material, and triggers data type conversions, thereby simplifying the complexity of applications. Contextual information is used to dynamically assist applications in finding relevant information. Novel features of the system include how the view of data adapts to the current context, how user data is imported into the local environment, and how the system adapts to device heterogeneity. We describe several applications that we have developed within our ubiquitous computing infrastructure and show how they leverage the novel features of our file system to simplify their complexity. The system is evaluated as part of a ubiquitous computing infrastructure deployed in a prototype environment to investigate issues of performance, scalability, and usability.
Acknowledgments

I have received a lot of support from many people during my stay at the University of Illinois and I am grateful for the help and encouragement they have provided me. Prof. Roy Campbell gave me the freedom to pursue interesting directions and was a great inspiration for new ideas. Prof. Klara Nahrstedt offered excellent advice during my research and I appreciate her sincere interest in her students. Prof. Dennis Mickunas and Prof. Robin Kravets were very helpful in providing feedback for this thesis and helping me recognize issues I may have not otherwise considered. The interactions I had with the whole Software Research Group were very valuable in terms of implementation and applications of my work. I would like to especially thank Manuel Român for many long discussions regarding the design of Gaia, applications of my research, and our talks about ubiquitous computing systems. Of course, I could not have completed by thesis without the help of Anda Ohlsson and Andrea Whitesell, who made many arrangements for me and made sure that things ran smoothly. I would also like to thank my family for much encouragement during this long process.
# Table of Contents

List of Tables ................................................................. ix

List of Figures ................................................................. x

List of Abbreviations .......................................................... xiii

Chapter 1 Introduction ......................................................... 1
  1.1 Systems Support for Ubiquitous Computing .......................... 2
  1.2 Interaction with Ubiquitous Computing Spaces ...................... 6
  1.3 Data Management Challenges .......................................... 7
    1.3.1 Shared Work Space ................................................ 7
    1.3.2 Presentation Room ................................................ 8
    1.3.3 Seminar Room .................................................... 9
  1.4 A File System for Ubiquitous Computing Spaces ..................... 10
  1.5 Thesis Contribution .................................................. 11

Chapter 2 Context .............................................................. 13
  2.1 What is Context? ....................................................... 13
  2.2 Challenges in Using Context .......................................... 15
  2.3 Location-Based Data Access ........................................... 18

Chapter 3 The Gaia File System ............................................. 20
  3.1 Overview .............................................................. 21
  3.2 Integrating Context .................................................. 22
  3.3 Data Administration .................................................. 26
  3.4 Architecture .......................................................... 27
    3.4.1 Mount Server .................................................... 28
    3.4.2 File Server ...................................................... 31
    3.4.3 Resolver ........................................................ 31

Chapter 4 File System Implementation ..................................... 33
  4.1 Layout Manager ....................................................... 33
    4.1.1 Layout Manager Interface ...................................... 35
  4.2 Containers ........................................................... 39
    4.2.1 Container Structure ........................................... 40
4.2.2 Container Interface ................................................. 41
4.3 Container Manager .................................................. 44
  4.3.1 Container Descriptions ......................................... 46
  4.3.2 Container Chains ................................................. 47
  4.3.3 Container Manager Interface ................................... 47
4.4 User Manager ......................................................... 50
4.5 Virtual File Hierarchy ................................................ 52
  4.5.1 Context Files .................................................... 53
  4.5.2 Context Directories ............................................. 53
  4.5.3 Directory Listing ............................................... 54
  4.5.4 Directory Creation ............................................. 56
  4.5.5 Attaching Context .............................................. 56
4.6 Prototype Environment ............................................... 58

Chapter 5 Gaia Operating System ....................................... 60
  5.1 Component Management Core ..................................... 61
  5.2 Event Manager ..................................................... 61
  5.3 Presence Service .................................................. 62
  5.4 Space Repository ................................................ 64
  5.5 Context Service .................................................. 65
  5.6 Security Services ................................................ 65
  5.7 Application Framework .......................................... 65
  5.8 Bootstrap ......................................................... 67
     5.8.1 Interface Repository ......................................... 69
     5.8.2 UOBHost ..................................................... 69
     5.8.3 Name Service ............................................... 70
     5.8.4 Other Services .............................................. 72
     5.8.5 Mobile Devices ............................................. 72

Chapter 6 File System Interface ........................................ 74
  6.1 Gaia Container Wrappers ......................................... 74
  6.2 Gaia File System Wrappers ....................................... 76
  6.3 Interface Usage .................................................. 77
     6.3.1 Attaching Context .......................................... 78
     6.3.2 Accessing Directories ....................................... 78
     6.3.3 Accessing Bytes ............................................ 80
     6.3.4 Format Conversion ........................................... 80
  6.4 File System Events ............................................... 82

Chapter 7 Applications .................................................. 83
  7.1 Gaia Data Browser ............................................... 83
  7.2 Attendance Recorder ............................................. 86
  7.3 Paper File List .................................................. 87
  7.4 Handheld GIF Viewer ............................................. 88
  7.5 Music Jukebox .................................................... 89

vi
Appendix A ................................................................. 147
  A.1 Container Interface .............................................. 147
  A.2 File System Interface ........................................... 149
  A.3 File Attributes, Modes, and Types ......................... 152

Vita ................................................................. 154
List of Tables

4.1 Example of several system (left) and user (right) mount descriptions. Entries which do not contain Mount tags are used in the construction of the virtual directory hierarchy. .............................................. 36

5.1 Kernel services that are started as part of the bootstrap process must be started in a particular order to ensure that the dependencies among services is preserved. ......................................................... 68

6.1 Catalog of implemented containers. .......................................................... 77

8.1 Measured times for context mode operations. ............................................ 101
8.2 Time in seconds to retrieve a GIF image object from a PowerPoint source. 102
8.3 Time in seconds to retrieve a Bitmap image object from a PowerPoint source. 103
List of Figures

1.1 The ubiquitous computing model moves applications away from a single machine with well known input and output methods to a space with many possible configurations. ................................................................. 4
1.2 Active spaces are populated by a variety of input and output devices. Applications are spread across the present devices and can be integrated into application components, such as lighting elements. .................. 5
3.1 The mount points of mobile users may be dynamically added to the space file system to make data available to applications running in the space. The system adds a temporary user directory under which personal mount points are added, which reference remote file servers. ................................. 23
3.2 An abridged view of the context mode virtual directory hierarchy. The virtual file hierarchy aggregates files and directories with the same context associated to them. Note that the time: context directory appears twice in the figure, illustrating that no fixed hierarchy is imposed on context. ............................ 25
3.3 The file system directs queries based on the current context. ............... 26
3.4 The architecture of CFS consists of a mount server (MS) that manages the layout of the namespace and file servers (FS) that provide access to distributed disks. The resolver directs file paths to file servers hosting native files. .... 28
3.5 The algorithm for finding valid mounts is used to to display relevant information in directories................................................................. 29
4.1 The structure of containers is composed of interfaces that define data object type and concrete implementations. The UML diagram shows the class hierarchy of the PowerPointContainer implementation. ...................... 41
4.2 The base container interface defines the generic methods and structures used by all derived container types. ............................................ 42
4.3 Derived containers define the data object type that a container holds. Read-only containers, such as the GIFContainer, only implement the getObjects method. ................................................................. 44
4.4 Servers manage their local native files and devices. The system can instantiate containers on any node in the system and adapt content for different device types. Rounded rectangles represent container instances. Hexagons represent template wrappers. ................................................................. 45
4.5 Example container descriptions for a file and converter container. ........ 46
4.6 The system constructs a map/graph data structure that is used to determine how to convert content to a desired type. .......................... 48
4.7 Container Managers instantiate and connect container chains. ............ 48
4.8 Container Managers enable adaptation of container interfaces. An existing container (a) can be converted to a new container type (b). ............ 49
4.9 Users may carry personal data references via handheld device and merge them into the local namespace. ....................................................... 51
4.10 The architecture of CFS consists of the mount server (MS) and file server (FS). The mount server defines the layout of storage for a space via mount entries, which contain pointers to storage on remote file servers. A mobile handheld can be used to carry personal mount points, which may be merged into the local namespace. When users enter without any device, the system can uses the authentication service (AS) and profile service (PS) to retrieve user mounts from a home server. .............................................. 52
4.11 An example user profile that contains the location of user mount points. .......................... 52
4.12 An example mount point generated when creating a virtual directory with location and situation context. ................................................. 56
4.13 An example mount point generated when attaching location and situation context to a file. .......................................................... 57
4.14 Context directories aggregate all files with the given context tags. Files may be owned by different users and may reside on remote file servers. Context mount points reference directories that contain files and links to data with the associated context. ....................................................... 58

5.1 *Gaia* provides core services for building ubiquitous applications. .............. 60
5.2 The Event Manager facilitates the manipulation of event channels. .............. 62
5.3 The Presence Service listens for the availability of physical and digital entities. 63
5.4 The Space Repository stores information about all active entities in an active space. .......................................................... 64
5.5 Each space is identified with a unique space id to bind a collection of services belonging to the same space. .............................................. 71

6.1 UML class diagram of the structure of the *ByteContainer*. The container base class is parameterized on *Container Type* (C) and *Object Type* (O). Template parameters are used to type data passed to the remote CORBA containers. 75
6.2 User-level containers are described using generic programming concepts to maximize core reuse. Typed objects are marshaled over the network. Groups of data objects can be cached in the local (generic) container. 76

7.1 The virtual file hierarchy is constructed from the meta-data tagged to files and directories. New meta-data can be added by creating new context directories and copying files into them. ....................................................... 84
7.2 User data can be added to the files visible to the browser. Custom application configurations may be added to the list of options when launching an application by appropriately tagging data. 86

xi
7.3 The attendance application uses the environmental context to read and write to the correct data source. ................................. 87
7.4 The file system dynamic data types allows applications to access data in the format that best suits their needs. ................................. 89
7.5 Files can be added to the local space by selecting a menu option from Windows Explorer. ............................................. 90
7.6 Files are retrieved from a laptop after a temporary mount pointing back to the laptop file server has been added to the mount server. .......... 91

8.1 The performance numbers were obtained from data from our active space that we use on a regular basis. ................................. 99
8.2 Response times increase linearly with the number of mounts. .......... 100

9.1 The seminar room contains a central conference table with wall mounted and table displays. .............................................. 105

11.1 An example mount point using the current design. ................. 124
11.2 An example mount point using an alternate design. ............... 125
List of Abbreviations

CAR  Context-Aware Retrieval
CFS  Context File System
CORBA  Common Object Request Broker Architecture
COTS  Commercial Off-The-Self
DCOM  Distributed Component Object Model
GAN  Geographic Area Network
GBP  Gaia Bootstrap Protocol
IDL  Interface Definition Language
IR  Infrared
GPS  Global Positioning System
GUI  Graphical User Interface
NA  Name Agent
NS  Name Server
ORB  Object Request Broker
PDA  Personal Digital Assistant
**PDF**  Portable Document Format

**RMI**  Remote Method Invocation

**SID**  Space Identifier

**SOAP**  Simple Object Access Protocol

**STL**  Standard Template Library

**UML**  Unified Modeling Language

**XML**  Extensible Markup Language
Chapter 1

Introduction

*Like the personal computer, ubiquitous computing will produce nothing new, but by making everything faster and easier to do, with less strain and fewer mental gymnastics, it will transform what is apparently possible.* - Mark Weiser

The original vision of ubiquitous computing was articulated over 10 years ago [Wei91]. The concept envisions computation becoming so cheap and small that it can be incorporated into everyday objects and becomes common place, moving to the “periphery” of human attention. Only recently have advances in communications, device miniaturization, and diverse sensors brought the goal of making computation pervasive closer to reality. The seminal research at PARC involved custom built systems, from wireless networking and handheld devices to software that tied the different parts of the system together. The goal was to augment the office environment to allow users to interact with installed computing systems from small handheld devices. A ground breaking concept was the incorporation of context into applications. Context can be anything from the physical world that affects an application, from location, time, weather, mood, etc. This was a radical departure from traditional applications at the time. Today we are starting to see contextual information making its way into real-world applications, location being the most common to date.

Today, ubiquitous computing has permeated almost all aspects of computer science. We see research involving hardware, sensors, networks, operating system, user interfaces,
software architecture, multimedia, and the list goes on. Beyond computer science, ubiquitous computing involves psychology, ethics, and human behavior; it has become clear that ubiquitous computing involves every aspect of science. In addition to the horizontal layers that ubiquitous computing covers, vertical cross sections have focused on specific domains. For example, an early adopter of ubiquitous computing technology is the automotive industry [Her02]. Automobiles are an ideal environment for ubiquitous computing, mainly because they are not constrained in the ways that other environments are; cars have less restrictions in space, power, connectivity, and cost. Already there are cars that people can converse with in order to get directions and find local restaurants or that can contact emergency services in case of airbag deployment [OnS]. Even within this domain alone, there are many different aspects of ubiquitous computing. For example, ubiquitous computing can refer to communicating devices within a car, between the outside world and the car, or among cars. In a nutshell, ubiquitous computing populates a huge space and is very broadly defined.

1.1 Systems Support for Ubiquitous Computing

Much of the focus of the ubiquitous computing community has been in the interactions between (wo)man and machine - forcing the machine to adapt to the user rather than the more common reverse approach. Natural language processing, interface design, gesture recognition, and so forth have received much attention in research circles with the goal of simplifying the use of the computational environment and resident applications for users. An oft overlooked area is the time spent developing these applications. The great success of the personal computer is largely due to the ease in which applications can be built and ported to new hardware developed by different vendors. This success is a result of the abstractions that the operating system provides to application developers. Recognizing this importance, an area of ubiquitous computing that has emerged recently is infrastructure support for ubiquitous applications. The initial approach in constructing ubiquitous applications was to custom
build each one to perform some specific function. It quickly became clear that many applications performed similar base operations and therefore required an underlying support system. For example, it became apparent that the acquisition of contextual information was a common requirement of many applications. From this evolved a toolkit that abstracted the contextual information obtained from surrounding sensors [SDA99]. As further experience was gained with constructing applications, other functions were found to cross-cut applications and were abstracted into reusable libraries or services that could be leveraged by a large number of applications.

Infrastructure support is necessary for building complex distributed computing systems. An area of current activity in ubiquitous computing research is bridging the virtual and physical worlds by augmenting physical objects with intelligent sensors [WFGH99, IU97, FIB95, GBK99] and incorporating an array of software, hardware, and physical entities into next generation distributed computing environments [Bro97, KB01, BMK+00]. These environments consist of intelligent rooms or spaces, containing appliances (whiteboard, video projectors, etc), powerful stationary computers, and mobile wireless handheld devices. There may be many input and output devices and non-conventional peripherals, such as lighting and temperature sensors, can be incorporated into applications, as shown in Figure 1.1. In effect, the surrounding physical space and its devices, peripherals, and appliances becomes the computational environment.

Such environments present many challenges in the development of applications [BBG+00]. What does it mean to move an application from the desktop to the surroundings? How are the applications constructed, how do users interact with them, and what is required from the support system to coordinate devices, peripherals, and application components? The heterogeneity, mobility, and sheer number of devices makes the system vastly more complex than traditional distributed systems. Applications may have the choice of input devices, such as mouse, pen, or finger; output devices, such as monitor, PDA screen, wall-mounted display, or speakers. It is clear that the infrastructure needs to locate available resources,
detect when new devices are spontaneously added to the system, support the mobility of
users, accommodate heterogeneous devices by adapting content when data formats are not
compatible with output devices, and export contextual information to applications.

![Diagram](image)

Figure 1.1: The ubiquitous computing model moves applications away from a single machine
with well known input and output methods to a space with many possible configurations.

The large collection of devices, resources, and peripherals must be coordinated and access
to them made simple. This coordination may be viewed as being analogous to the role of a
traditional operating system, but extended outside the confines of the desktop to encompass
the surrounding devices in the local environment. Many of the existing operating system
concepts can be naturally extended into the domain of rooms [RHC+02]. For example, event
systems and tuple spaces have been identified as important services to provide decoupled
communication among system components in ubiquitous computing environments [KF02,
JF02]. These events and tuples are similar to signals and shared memory in desktop systems.
However, relevant issues in ubiquitous computing require that the supporting middleware be
extended to accommodate this new operating environment. In addition, the system should
be able to incorporate legacy and commodity off-the-shelf (COTS) components, while at the
same time making it easy to build new applications. Construction of such a (meta-level)
operating system can be achieved through the use of middleware in order to leverage the
vast array of existing applications, native operating systems, and devices.
We term these physical environments and the software used to coordinate them *active spaces* (see Fig. 1.2). Active spaces are often bound by the physical environment, such as the walls in a room, but are not restricted in this way. For example, two or more rooms may be joined to comprise one active space, or a single room may be split into more than one active space.

![Diagram of active space components](image)

Figure 1.2: Active spaces are populated by a variety of input and output devices. Applications are spread across the present devices and can be integrated into application components, such as lighting elements.

Active spaces cover a range of environment types, including offices, workspaces, classrooms, and homes, and have several defining characteristics, which include the following:

- **Mobile.** Users and their devices are mobile. Users may possess handheld devices, which can be used to interact with the resources and applications resident in the physical space and these devices can become integrated with the other resources of the space. Users can move between spaces and their environment (i.e., applications, state, data, etc.) can move with them.

- **Heterogeneous.** A wide variety of devices exist, which exhibit different resource requirements and form factors. Application can be mapped onto different spaces depending
on the available resources.

- **Interactive.** Applications running in the space are interactive, may be controlled remotely, and may be notified of event changes that take place in the space or applications.

- **Networked.** The software components running in the physical space are connected through a wired network that supports multicast to discover services within the space. Mobile devices are able to interact with the services and applications in a space through wireless networking, such as IEEE 802.11 and infrared.

- **Distributed.** Applications may utilize any distributed resource within a space and may be partitioned to increase the amount of information presented to the user. Portions of an application may also be moved or duplicated on different devices.

### 1.2 Interaction with Ubiquitous Computing Spaces

The ways in which ubiquitous computing spaces are used differs substantially from traditional desktop systems. They can be used as shared work spaces or to support group activity. There is no longer a one-to-one relationship between users and applications. Typical desktop applications are monolithic entities that are launched and run on a single machine by double clicking a data file or application icon. Space-hosted applications can be scheduled to execute at a prescribed time, may have minimal user interactions, can be configured to run on different individual devices, or can be spread across the available computing nodes.

When supporting group activity, applications are often not run as isolated entities, but as collections of software components that constitute a task [WG00]. A task may involve one or more people and may last for a certain duration of time. Shared work spaces often include continually running applications whose lifetime is greater than the duration of time a user occupies a space. Users must be able to interact with such applications for a short length
of time, perhaps adding personal data into them, and then leave without the application being terminated. Users may possess mobile personal devices that they carry with them into such environments that can become a part of the computing environment and can be used to control or host application components. Applications can be launched using default configurations based on what resources are permanently resident in the space or can be customized through configurations that utilize a user's personal devices.

The software running in such a space must be able to support all of the above uses. These directly affect the structure of applications and the way in which the data they use is introduced into the space and integrated into them.

1.3 Data Management Challenges

The ways in which ubiquitous computing spaces are used places new requirements on the data management system, which is a result of several factors; users are mobile and can inject data into the local environment, applications may be required to alter the format of data to support different device capabilities, user-defined configurations may be used to specify the layout of applications, and users can direct portions of their personal data to different application that are already running. To make the data management challenges more concrete, we give three motivating scenarios that describe how an active space can be used and we then highlight the data management issues that are raised by each.

1.3.1 Shared Work Space

Scenario. A user enters with a laptop into an active space that is populated by other workers. A shared music jukebox is running in the space that plays music through the room speakers. After being detected by the space, the user selects some personal music files from the laptop and adds them to the space. The jukebox songlist is automatically updated to include the new files, which are pulled from the laptop when played. When the user leaves
the space, the files are removed from the songlist.

**Issues.** A user may enter a space that has several applications that are already running, to which they may wish to add some information. However, how does each application know what data is meant for it if the data is dynamically added after it has been launched? Each application may accept a certain type of data; for example, a jukebox accepts music files and a file browser accepts application configurations with which to launch applications. The applications must be notified that new files are available and must be able to determine which are the meant for it.

### 1.3.2 Presentation Room

**Scenario.** A user enters a conference room with a PDA and wishes to give a presentation. The room detects the presence of the user and their personal data and preferences become available to the space. The user walks to one of the wall mounted displays, finds the personal data from a file browser that is running in the space and selects a personal configuration that runs the presentation application on two displays and places a control component on the personal device. One of the participants is late and would like to start viewing the presentation on their handheld device on their way to the conference room so that they don’t miss the beginning of the talk.

**Issues.** Users are highly mobile in active spaces and should not be burdened with manually transferring files or data, be it configurations, preferences, or application data from one environment to another. The environment can assist in making personal storage automatically available in the users’ present location. Storage becomes implicitly linked to a user and can “follow” them around, becoming available to applications whenever they enter a new space. Users become the center of their personal *data cloud*. This cloud contains all the data that is allowed to be accessed in spaces that the user may visit.
Due to the large number of available resources, applications may be configured differently based on what resources are available or on how a user prefers to use such a space. Therefore, an issue that arises is how the user-defined configurations can become part of the choices available when launching applications. A user may wish to launch a presentation on several wall mounted displays for formal presentations, or alternately use available personal devices for more casual interactions. When users introduce new devices into a space, the number of possible configurations grows if the device can host application components. These configurations can incorporate mobile devices that are not known a priori to the space, but may be known by users if they include their own devices, for example. In addition, configurations may be specific to certain locations (i.e., a presentation may be configured to run on several fixed displays in a certain space). Therefore, when choosing from an list of available configurations, only the relevant ones should appear. For example, if a user is in room 2401, configurations for room 2402 should not appear as possible choices.

A further issue is how application data can adapt to heterogeneous devices. Active space environments are typically populated by a wide variety of computing devices. Applications may no longer make assumptions about the types and characteristics of devices that a user may possess or that are resident in a particular space. If a user wishes to use a device that does not support the original data format or prefers to receive some data in a different format, the infrastructure must make an attempt to present the data in the desired format. Applications should not be bothered with the complexities of such conversions; they should gain access to data in a particular format by simply opening the data source as the desired type and the system should be responsible for automatically adapting content to the desired format.

1.3.3 Seminar Room

Scenario. A group of people gather in a seminar room to discuss some papers. This group activity has been scheduled to begin at a certain time each week and automatically launches
several applications. When the seminar begins, the papers for the week are loaded into a file viewer so that when selected, they are displayed on the large wall monitors. In addition, an attendance recorder is also started. Once users are detected and identified by the space, the attendance application records the participants name and timestamp onto secondary storage so that attendance can be tallied at the end of the semester.

**Issues.** The issue raised by this scenario is how the application data is found when certain applications may be automatically launched with minimal or no ensuing human intervention. An application may know what type of data it requires, but may not know where it is located. For example, the attendance recorder may be used for different activities, such as a meeting or seminar, but since it is launched by the system, has to find the source of saved data. By using the current context, data can be associated to an application and activity, thereby making the data available whenever the context is active.

### 1.4 A File System for Ubiquitous Computing Spaces

Based on our experiences using a ubiquitous computing space, we found that existing data access mechanisms did not accommodate the types of applications that we found to be useful in such spaces, and the ways in which people interact with those applications. No existing file system has made use of different types of contextual information to support ubiquitous applications by assisting them in the location of data, adaption of data, and availability of data.

To address the foregoing issues, this thesis presents a file system targeted at ubiquitous computing spaces and supports the requirements imposed by the environment. The file system is designed to support the different types of applications and devices that ubiquitous computing spaces introduce, and enables:

- mobile users to make their data available to the local space
• context to define what information is important for an activity
• automatically launched applications to find data
• users to inject data into running applications
• users to import application configurations into specific spaces
• data to be accessed in different formats

The design of the system is a combination of a database and file system. Each space maintains a namespace into which users can inject references to their personal data. The system maintains a database that applications can query to find where relevant material is located and is updated with user information when they make their data available to a space. The query mechanism is hidden behind a file system abstraction; applications simply open target directories to find pertinent data, the contents of which may change as context changes and is limited to the data that the application requires. They then receive updates when new information has been added or removed as users enter and leave a space.

Our file system is implemented at application level, which offers several advantages. From a developer viewpoint, the system is more easily implemented and debugged. From a user point of view, an application-level approach allows users to create and manipulate data from existing applications that can be subsequently used in the ubiquitous environment. For example, users are able to create and use native system files with standard applications, e.g., office productivity tools. This data is then accessible from custom active space applications by using the file system interface. In this manner, users may use familiar applications while on their personal desktop machines and then use any generated data from an active space.

1.5 Thesis Contribution

The design, implementation, and evaluation of the file system presented in this thesis contributes to the current research in ubiquitous computing by explicitly integrating context
into its design. By making the file system context-aware, new types of applications are supported, user data is made available to the local environment, and data is allowed to adapt to device heterogeneity. The implementation and subsequent experience with the system has contributed an increased understanding in how data management systems must be changed to accommodate the unique characteristics of ubiquitous computing.

The context-aware file system, the focus of this thesis, is an integral part of a distributed operating system called Gaia. Gaia extends the traditional concept of the operating system to the domain of rooms by providing an abstraction layer for building applications that make use of the many resources that are available in the local environment. This abstraction layer allows applications to be developed in a generic manner that are able to run in different environments with different resources. While the specific concept studied in this thesis is the file system, the other parts of Gaia are discussed, as the file system relies on other components of the system.

The remainder of this thesis is organized as follows: Chapter 3 describes the architectural design of the context-aware file system, including the mount server, file server, and resolver. Chapter 4 explains the implementation of the system and Chapter 5 describes the remaining services that Gaia provides to ubiquitous applications. Next, Chapter 6 gives more details regarding the application programming interface, Chapter 7 describes some applications that we have developed and how they leverage the file system features. Chapter 8 evaluates the system in terms of performance and usability and Chapter 9 describes how we have used the system during our weekly seminars and what lessons we have learned through routine use. The thesis finishes with related work in Chapter 10, future work in Chapter 11, and concluding remarks in Chapter 12.
Chapter 2

Context

The research community has tackled the many problems in ubiquitous computing from different directions. We took a systems approach to ubiquitous environments and were interested in how this environment would affect the design of distributed system and what new services would be required to support ubiquitous applications. While many concepts that arise in ubiquitous computing are not unique to the environment, such as mobility, the ways in which these concepts must be supported differs because of the ways in which users interact with the environment. Other issues, such as the integration of context, are significant departures from typical distributed systems. Contextual information provides applications with a richer set of environmental attributes so that they can adjust behavior in some way or trigger actions. The file system that we present in this thesis makes use of contextual information to simplify application development and to enable new features. In this chapter, we give a description of what “context” means and then explain how context can be used in data management systems.

2.1 What is Context?

Studying the use of context in applications and systems is a key contribution of ubiquitous computing research. So what exactly is context? Dey et al. [DAS99] defines context as “any information that can be used to characterize the situation of an entity, where an entity can
be a person, place, or physical or computational object.”. Contextual information therefore is environmental information that is used to help describe something through interpretation [Win01]. For example, “come here” means different things based on where the phrase was uttered; that is “here” is described by the current location. Some concrete examples of context are:

- **Location** - represents the location of the current space, such as a specific room number
- **Identity** - is the distinguishing character of an entity, such as a person, device, or object
- **Activity** - refers to an activity that is taking place within a space, for example a meeting or lecture
- **Space** - represents the type of space, e.g., shared, private
- **Time** - can specify a valid duration for data, e.g., one week
- **Device** - can specify the characteristics of a devices, e.g., graphical context

The goal of context-aware computing is to make the interactions between users and machines more natural and to make such systems easier to use [DA99]. Context information can be used in “the presentation of information and services to a user, automatic execution of a service, and tagging of context to information for later retrieval” [DAB⁺00]. Therefore, context can be used to prepare an environment for activities, to adapt an application to the current surroundings to better suite the needs of the user, and to provide relevant information to applications and users based on the current task. Contextual information is very much a part of the physical environment and therefore including contextual information into computing systems results in an integration of the physical and digital environments. However, this integration is by no means trivial and there are many challenges so that the resulting marriage is meaningful and useful.
Integrating context information into computing systems has the potential to greatly simplify the interactions between users and the environment. For example, consider a person working in an active space such as an office, which is populated by a high quality audio system. The user may bring with them a laptop, but wishes to use the room speaker with their superior sound quality. The data (i.e., music files) can be exported from the laptop to the space so that they are available to a jukebox application running in the space. A part of the contextual information describes the space as private, since only one person is present. However, if another person enters the space, the context has changed. It moves from a private space to a shared space and the application may react to this change in context by moving the audio output back to the laptop. A similar situation could be envisioned for documents; in the above example, replace the audio with personal documents and the speakers with public displays. Again, the fact that the space changes to a public (i.e., shared) may trigger a change in application behavior. If the user then moves from the office space to an automobile, the context again changes to include “driving”. Now if a user runs a calendar application in the car, the change of context may require the application to reconfigure itself so that input and output are performed with the audio system in the car rather than with mouse and display because “driving” requires minimal distraction. Context therefore becomes important in the dynamic configuration of applications and the surrounding environment.

2.2 Challenges in Using Context

One of the challenges in integrating context into computing systems is in accurately reporting contextual information and to have the application do the “right thing”. Some ubiquitous systems strive to be proactive [SG02, GSSS02], that is they react in some way to benefit the user or anticipate future actions to prepare the environment. Such systems must make sure that the actions performed are what the user desires or is expecting. When a system takes
actions based on incorrect context information, proactive operations become an annoyance and the purpose is defeated and therefore the use of contextual information must be carefully considered. To date, only simple types of context information have been integrated into ubiquitous systems, which attests to the difficulty of the task. Many such systems require unambiguous information in order for the system to perform with deterministic behavior. However, inferring more complex context from environmental sensors is difficult and even simple deductions "cross the line between 'regular' technical systems and systems that require human-like capabilities" [Leu02]. Context awareness quickly becomes a hard artificial intelligence problem, one that has not yet been solved [Dry01].

In dealing with context, several issues must be addressed. The first is how context information is gathered from the environment and presented to applications. Early work employed dynamic environment variables that changed value as context changed. Applications received callbacks when the value of any variable was altered and the application could then choose to adapt behavior [STW93]. It was soon recognized that there was a need for a reusable system by which contextual information could be acquired by applications through reusable services [Pas98]. Following work (i.e., the Context Toolkit [SDA99, DAS99]) represented context as widgets and presented context as building blocks from which applications could be built. The widget abstraction is akin to the GUI widgets, where applications can pick and choose from a choice of available context items to suit their needs. Widgets typically encapsulate some physical sensor, such as a temperature sensor, and act like a device driver. However, they can also be implemented on top of a network service, such as a stock price service. More complicated context information was represented by aggregators that included two or more widgets which could be combined to infer some more complex contextual information. The canonical example is an aggregator that determines when a meeting is taking place based on audio levels and number of people in a room. When the number of people becomes greater than a certain threshold and the audio level rises above a given level, the aggregator concludes that a meeting is in progress. The toolkit approach
has some limitations and others have argued for a network service model [HL01, HMH02] to overcome some of these limitations. Benefits of this model are increased reusability, dynamic context interpretation, and distributed processing. Other researchers are developing formal models which define entity relations on which more complex logic operations can be performed [HIR02, RCRM02].

The second issue is how applications receive context information. Several methods are common across most of the available context gathering systems. One method is to query the system for information, i.e., an application may query for the local temperature. This method is especially useful to determine context history, that is context information that was valid in the past and may or may not still be valid, e.g., an application may want to know if a certain person was in the room that day. A second method allows applications to subscribe to context information. These systems are built using the publisher/subscriber paradigm, where context producers publish context information and context consumers register for interested context information. When the context subscription becomes true, registered consumers are notified, typically via a callback or an event distribution mechanism. Much research has gone into how to efficiently route and match subscriptions to valid conditions [ST94, ASS+99, SAB+00].

A third important issue is how applications ask for context information. An application is usually interested in some specific type of context information to which it reacts. Not only does it have to know if the environment supports the context it desires, but the application also has to know how to ask for it. Many systems define context properties through tuples. In this case, the application must know what keywords to use when describing interested contextual information. Ontologies have become a popular tool for describing context information [SBH02]. Mapping between different ontologies with similar semantics is also being investigated to bridge different systems.
2.3 Location-Based Data Access

The most widely used piece of context information (commercially and in research) is location. Providing services based on location is useful in narrowing the amount of relevant information to what is available in the local environment. Some of the first context-aware applications were tour guides [AAH+97, LKRF99, DCME01] that used the location of the user to display information about nearby buildings, attractions, or landmarks, which automatically changed as the person moved. Another deployed system allowed a user to bookmark interesting information in the physical environment from a mobile device so that the information could be later retrieved [FFK+02].

When using location in data retrieval, the main idea is to limit the data to the current proximity, which is generally more relevant. Consider a person at a bus stop. They may want to know what restaurants are nearby or when the next bus is scheduled to stop at the nearest station. In this case, restaurants not in the local vicinity would not be included in the results. Therefore, the information is scoped [Hew]; information that is “further” away moves to the periphery and “closer” information is made more readily available.

Location has been one of the first pieces of context information adopted by applications because it can be obtained reliability with a good degree of accuracy. Several web sites provide weather forecast information for the town where the user is accessing the web page [Theb]. Current practice on the web allows users to enter the local zip code that the browser saves in a cookie. Web requests to the web server pass the zip code to find the local weather information. However, while useful, this method is limited in that it requires the user to enter information (which may not be known) and is static, assuming that the user’s computer is in a fixed location and does not move from the local area. However, for mobile applications, this assumption does not hold. Users may be moving and the current location must be frequently updated to keep information relevant. Context is a dynamic property that an application must continually monitor in order to present the correct information to
the user. For mobile users, the location of the user can be tracked via GPS, active badge, beacons.

A wide range of contextual information, other than location, can be used to help applications locate pertinent material. For example, the activity taking place in a room may dictate what information applications should use. In the next chapter, we describe how our data management system integrates context information to support the activities that take place in a ubiquitous computing space.
Chapter 3

The Gaia File System

Our approach to ubiquitous computing has been to extend the concept of the operating system into the room environment. The goal was to treat a physical space with its local devices, peripherals, and appliances as one large programmable computer. However, in this environment simply starting an application is a challenge, not to mention moving or partitioning an application or accessing data. To achieve these objectives, we developed Gaia, a middleware meta-level operating system, of which more details are given in Chapter 5. In developing this ubiquitous infrastructure for active spaces, it became clear that a file access mechanism was required. Since we were abstracting the space and its devices as an operating system, the logical choice was to use the file system abstraction. However, could we use an existing file system design or was it necessary to develop something new? In essence, was there a need for another file system? What was new in this environment that required added support in regard to data access? Active spaces embody a new interaction style among users, devices, systems, and applications. Much of the way in which these interactions are performed are dictated by contextual information and therefore context became a driving factor in the design of the file system and in what features it was required to support.

The Gaia file system is context-aware; that is, it is sensitive to the location of users, the context in which an application is running, and the context of the devices using it. The inclusion of context sensitivity allows the system to assist applications and users in a ubiquitous computing environment and is used to alleviate many of the tasks that are traditionally
performed manually or require additional programming effort. More specifically, context is used to 1) automatically make personal storage available to applications, conditioned on user presence, 2) organize data to simplify locating data important for applications and users, and 3) retrieve data in a format based on the context of user preferences or device characteristics. Our goals in developing the system were to investigate how context information could be integrated into a data management system, what types of context would be most useful to support useful functions, and how best to support application development through appropriate application programming interfaces.

3.1 Overview

Our file system, called the Context File System (CFS), categorizes context into external context and internal context. External context is defined as any information that is gathered from the surroundings, outside the scope of the current device or application. Internal context is defined as any information that is determined from the current device or application, for example, device characteristics (i.e., graphic context) or user preferences such as data format.

External context is used in two ways. First, the location of the user is used to trigger the addition of personal data references into the current environment, thereby making user data available to local applications. Users can specify what portion of their personal data should be exported to remote spaces and it is dynamically added when they are detected within a space.\(^1\) The second use of external context is in the organization of application data. A context-driven view of data is used to organize information by limiting the visibility of files and directories to what is important within the current context (i.e., task). The system allows files and directories\(^2\) to be tagged with metadata (as a list of type/value tuples), which may be user or application-defined properties (i.e., type, subtype, category) or may

---

\(^1\)The presence of a user may be detected in several ways, including finger print matching, RF badges, cameras, electronic rings, etc.

\(^2\)For the remainder of the thesis, files will refer to both files or directories.
be contextual information defined by the environment. We call both properties and context attributes of the data. Properties are used by applications to determine what information they are interested in; context is used to determine when that data is made available to the applications. Applications access data by specifying the properties they are interested in (e.g., a music jukebox is interested in music files) and the system uses the current context to determine what portion of the data is relevant. The metadata is converted into a directory representation so that applications can easily access data. For example, in our seminar scenario described previously in Chapter 1, the PDF reader application can simply open a directory representing papers. The file system uses the current location, situation, and time information along with the fact that papers are requested to find the correct files for the application. The contents of the directory containing the resulting files automatically changes every week as papers are added and old papers time out. However, from the application point of view, it simply opens the same directory every week and finds the relevant material. Therefore, context determines what files are available in a particular directory, which changes as the current activity taking place in a space changes. By separating the context knowledge from applications and placing it in the infrastructure, applications may run unchanged in different situations.

Internal context is used to support heterogeneous devices and user preferences through the use of dynamic types. Since some devices may be limited in how they may present data to a user or a user may wish to receive data in a certain format (e.g., audio), the system allows applications to open a data source as a desired type. The system attempts to convert the data to the requested format if the appropriate (set of) transformation(s) is available.

3.2 Integrating Context

Our system uses the concept of the traditional mounting mechanism employed by many file systems to specify the placement of data in the directory hierarchy. Each space maintains a
collection of data references that constitutes the *space file system*, which consists of space-specific (system) data and remotely-located personal (user) data. Users maintain personal *mobile mount points* that may be merged into the space file system to make their data available within the space and act as pointers to remote storage, as shown in Fig. 3.1. The personal storage of users is dynamically added under the directory `/users/<username>` when they are detected within a space causing the namespace to change as users physically move in and out of a space. The file system operates in two modes. *File mode* makes all data that users have exported to a space accessible under the allocated *user* directory. An alternate view of data that is context-driven is used to organize data by limiting the visibility to what is important for the current context, which we call *context mode*.

![Diagram](image)

**Figure 3.1:** The mount points of mobile users may be dynamically added to the space file system to make data available to applications running in the space. The system adds a temporary *user* directory under which personal mount points are added, which reference remote file servers.

In *context mode*, the metadata attributes, composed of types and values, are used to create a virtual hierarchy. Paths are composed of these type/value pairs in the form `/<type>/<value>/*`. Material with the same type/value pairs (i.e., task relevance) are ag-
ggregated into the same directory, which may reside on different file stores owned by different users. The reasons for choosing this syntax are threefold: first, the syntax is simple and limited to the functionality we require; second, it allows our applications to use the same API for reading regular and context-aware files. Files are tagged with specific metadata by copying them from file mode into a virtual directory in context mode representing some attributes (i.e., sequence of tuples). Once attributes are associated to a file, the data is visible in the directory representing the context tuples, as shown in Fig. 3.2. This simple mechanism for associating meta-data to files results in the third reason; with the syntax, we were able to build a graphical way in which to manipulate context directories. A syntax which entailed complex queries would be much more difficult to operate graphically.

Since files may be aggregated from several locations and made to look as if they are in the same directory, there exists a possibility of name clashes. To avoid this possibility, all files and directories have an associated index. When files are accessed, this index is used to disambiguate files with the same name.

The virtual directory hierarchy forms a simple query language that can be used to determine what attributes are attached to files. The path of the directory is used to identify the attributes of interest. For example, to determine which files have the tuples $\text{location} == 2401 \&\& \text{situation} == \text{meeting}$ attached to them, one may enter the $/\text{location}:2401/\text{situation}:/\text{meeting}$ directory (location and situation can be defined by the context of an activity). It is important to note that although the context directory structure is viewed as a hierarchy, context directories impose no fixed ordering, resulting in a forest rather than a tree structure. While the above example illustrates that queries naturally support $\text{AND}$ boolean operations, $\text{OR}$ queries are supported by attaching different contexts to the same file.

Appending the special keyword $\text{current}$: to a directory path instructs the system to show only the information that pertains to the current activity by using the context of the environment (e.g., location, time, situation, weather) together with the application specified
properties in the path to display the correct data. Once data has been appropriately tagged, the system directs applications to the correct data based on the current environmental context information. For example, consider the case where a PDF viewer is automatically launched and must find the relevant papers to display. The application requires PDF files, so it is interested in data tagged with type == pdf; type is an application defined property. However, depending on the current context, different papers are relevant, and the system figures out which ones those are on behalf of the application. Therefore, the PDF viewer application can simply open a directory for the current papers, e.g., /type:/pdf/current:. The contents of this directory may automatically change every week, as papers are added and old papers time out. However, from the application point of view, it simply opens the same directory every week and finds the relevant material. Figure 3.3 illustrates the operation. Circles are PDF files, but only some of these files are meant for a particular seminar that takes place in a certain location and time, whose data is specified by the dark shaded shapes.
Therefore, an application wanting to view papers will only see the dark circles, which repre-
sent the papers for the context. However, it is important to note that the application is
ignorant of the current context; it simply opens a directory representing the “current” pa-
pers. The same application may run in different contexts - unchanged - and finds whatever
information is pertinent for the current context.

![Diagram](image)

Figure 3.3: The file system directs queries based on the current context.

### 3.3 Data Administration

The use of properties and context in the file system specify an agreement between applications
and users as to what data is important and when it should be made available. Therefore,
in using the system, different responsibilities fall on the application developer and space administrator.

Applications typically handle a certain kind of data. In context mode, the path of the directory or file can be used to specify what kind of data the application is interested in. For example, if an application opens `/type:/mp3/current:`, it has specified that it is interested in files of type `mp3`. Alternately, opening `/catagory:/music/current:` specifies that
the application is interested in any type of music file. Therefore, the application developer defines what kind of data the application is concerned with and in what directory to find the data.

System administrators and users tag the data so that they can be used by applications running in the space during specific contexts. They must know what the available contexts are and when they will be valid. For example, consider an attendance recorder that stores the attendees during an activity, such as a meeting. When the application starts, it opens a particular data file in a context mode directory. The application developer chooses the name of the data file. However, since the application can run in different contexts, such as seminars and classrooms, several attendance files may exist, only one of which is active at any time. The administrator or user must tag the data so that it matches a potential context so that the correct file is presented to the application.

Therefore, an administrator or user must know both the data properties that an application is expecting and also the contexts in which the application will run. Currently, these attributes are well known since there are only a limited number of applications and activities that exist. However, as more applications are developed and more activities are scheduled for a particular space, a way for these values to be advertised will be necessary and is an area of future research.

3.4 Architecture

In this section, we describe the architectural design of our implementation and describe how context is integrated into the file system components. The architecture is composed of mount servers that manage the layout of the namespace, file servers that provide access to distributed disks, and a resolver that parses file paths and determines which file servers host data and what the native file name is. The file system is a hybrid between a database and a file system; the database functionality offers the flexibility to search for relevant information
and the file system functionality provides an interface that application developers are familiar with. The mount servers implement the database functionality and are queried to find what files are associated to properties and contextual information. The file servers implement the file system functionality and provide an interface for accessing remote files (see Fig. 3.4).

![Diagram of CFS architecture](image)

Figure 3.4: The architecture of CFS consists of a mount server (MS) that manages the layout of the namespace and file servers (FS) that provide access to distributed disks. The resolver directs file paths to file servers hosting native files.

### 3.4.1 Mount Server

Each space maintains a single mount server, which stores the current storage namespace layout of the space file system. The mount server contains a collection of mount points, which are small data structures that reference data located on different machines. Mount points contain information that is used to determine how the referenced data fits into the namespace, where the data is located, and who owns the data. Our mount points have been augmented to contain the meta-data tags corresponding to properties and environmental context. We separate the meta-data from the actual data so that the meta-data can be easily searched, with only a minimal amount of information required to be transported as users move between spaces. The underlying data is stored as native files, since most existing applications use files as a source of data. The mount server may be searched for mount points
that reference relevant information based on a desired sequence of type/value tuples. Path resolution is performed by obtaining the mount(s) that correspond to the path components, extracting the native path(s) exported by file server(s) hosting the data, and resolving the path into native path format. When resolving the regular path syntax (i.e., file mode), a single mount is returned based on what path prefix matches an entry in the mount server. Context mode path resolution may return multiple mounts for a given tuple sequence, e.g., data residing in different locations may be tagged with the same context meta-data, thereby providing data aggregation. Mount points are described in XML format and are described in more detail in Chapter 4.

```plaintext
1  ctx   = array {type, value}
2  mount = {host, path, mount, ctx}
3
4  c.ctx = current environmental context
5  q.ctx = query context or properties
6
7  foreach (m in mounts)
8    if (q.ctx == m.ctx)
9      table1 <- m
10
11  foreach (m in table1)
12    foreach (c in c.ctx)
13      if (c memberof m.ctx ||
14         c.type not existin m.ctx.{types})
15      table2 <- m
16
17  return table2
```

Figure 3.5: The algorithm for finding valid mounts is used to display relevant information in directories.

In order for queries to be processed correctly, the mount server maintains a view of the current context, which is used when requests are interested in files in a current: directory. While accurate collection of context information is difficult, we have employed a simple approach in which the context for a certain duration of time is set via a calendar application, an approach that works well for our objectives. First, we don’t want context changing too often.
Second, we don’t want inaccuracies in the context that the system is using. As described previously, proactive systems deteriorate quickly when context is not accurately reported. Since ubiquitous computing environments support activities (information about which is already available within the calendar), tying the scheduled activities into the computing environment was a reasonable way in which to set the contextual information [MT01]. Setting contextual information in this way makes the system much more deterministic, predictable, and practical. A context service is currently under development that will be responsible for gathering contextual information from sensors and delivering this information to the mount server. In this case, the calendar will become an activity sensor.

The mount server responds to context mode path queries based on the tuples in the path and the environmental context and returns valid mount points that satisfy the query. The algorithm for handling mount queries is shown in Fig. 3.5. The mount server first finds the mounts that contain meta-data tags that exactly match the query tuples. If the keyword current: is included in the query, the current environmental context is then used to narrow the result to those mounts in which each tuple either exactly matches the mount tags or the mount does not contain a tag of the tuple type. The reasoning behind this is as follows: if the environmental context defines some value that a file has not been tagged with, that context is not considered an important attribute of the file and acts as a wildcard. Therefore, if the environmental context defines some attribute that is not meaningful to a piece of data, the mount point is returned as a result. For example, if the environmental context defines “situation” as an attribute, but an MP3 file is only tagged with a temporal property, the mount is valid, essentially making the MP3 available regardless of what the situation is. However, if the MP3 is tagged with a situation attribute, it will only be visible if the specified situation is active. This restricts the environmental context from hiding the visibility of data for which the context is irrelevant, since when data is tagged with meta-data properties, the full collection of environmental properties that are present in a space may not be known.
3.4.2 File Server

File servers manage data on the machine on which they are executing and use the low-level operating system to access disk. Data is strongly typed and can be accessed in different formats based on the type that it is opened as, resulting in a system that is *dynamically typed*. Since different devices have different characteristics and users may have preferred ways to represent data (e.g., text v. audio), dynamic types support device heterogeneity and resource availability. Software written for a particular device is typically aware of the device capabilities and therefore data can be accessed in a form that best meets the requirements of the device. Dynamic types are realized through the use of modules, which contain the logic to handle the structure of a particular type to facilitate access to the underlying file format or to convert between formats [HCM01]. There exists modules to represent various image formats, presentations, directories, news items, etc. The system is able to convert the type of the stored data to the desired type requested by an application by finding the correct module or modules to handle the conversion (if available). For example, a presentation file may be opened as bytes, GIF images, pixmaps, or bitmaps. However, some files do not contain any well-defined structure. Such files may be represented as a stream of bytes, thereby supporting traditional file system operation. Some modules support specific attributes that may be used to affect the data type in some way, such as changing the dimensions of a retrieved image object. These attributes allow for finer grained control over the details of the data characteristics once the coarse grained format has been chosen.

3.4.3 Resolver

The resolver is part of a client-side library that is linked in with applications. The library stores no per application state; state is stored in a small library header file, which is compiled into each application. This state contains information such as module attributes or the

---

3 We reuse existing applications by wrapping them with our application framework [RC02].
current working directory. The resolver maintains a cache of references to machines exporting
storage and provides file mode name resolution facilities. The resolver includes a prefix table
mechanism [WO86] and, when needed, attempts to make connections to available remote
file servers listed in the prefix table. The prefix table cache is initialized from the available
mounts in the mount server. The resolver maps UNIX-style directory formats to the native
machine format. For example, a directory /scratch/junk could be mapped to C:\Temp\junk
on a Windows host.
Chapter 4

File System Implementation

In this chapter, we detail the management of the namespace, the mechanism to perform data format transformations, how data sources are represented, and how remote disks are accessed. Mount and file servers are implemented as middleware services and utilize the native operating system to access disk. The file system components are implemented as CORBA [The98] objects with interfaces to manipulate them. We describe the implementation of each of the main components, along with several of the important methods available to interact with each component. In the next chapter, the high-level programming interface is presented that conceals much of the low-level file system details.

4.1 Layout Manager

The mount server manages the layout of the namespace and is a database of mount points (called the mount table) that may be queried to locate information associated to a particular path. The mount server is implemented as a component called the Layout Manager. The Layout Manager contains two types of mount points storage mappings; static and dynamic. The static mappings refer to storage that is always available to a space. This may consist of local storage or shared remote storage. The dynamic portion changes as users move between spaces resulting in new mount points being added or removed from the database. Users specify which storage they wish to export to other spaces by adding mount points to their
personal profile. By merging these mount points into the Layout Manager, the namespace gets updated and the data referenced in the mounts are made available to the space. The Layout Manager is initialized with an XML configuration file, which contains the space-specific static system mounts. This file contains mount entries that specify which machines export a part of their storage, how that storage gets mapped into the space file system namespace, to whom the descriptions belong, and (optionally) what context is associated to the data being referenced. Each XML mount point entry can contain up to five fields, which are:

- **Owner.** Defines the user name of the owner of the mount. This user name is unique within the system.

- **Mount.** Specifies where in the directory hierarchy that the referenced data will be placed. A mount must contain only one path component i.e., only a leading slash.

- **Host.** Specifies the host machine on which the data is stored.

- **Path.** Determines the location of the directory on the native host.

- **Context.** Specifies the attributes (meta-data) that are associated to the data referenced in the mount. Any number of context tags may be contained within a mount description, each of which is defined by a type/value pair.

Table 4.1 shows five example mount point descriptions. The first description is a system mapping that specifies that machine *pc2401-1* is exporting its *C:\Temp* directory and it will get mapped to */scratch*.

1 The second mount is a temporary directory that represents the context *situation == meeting && location == 2401* and has been generated for the local space (system). The remaining three mounts (right) have been dynamically added by users. The first of these specifies a mount that gets add to the namespace as */users/ckhess/office* from a remote machine. The last two mount points are references for two different users that

---

1The user name *system* is reserved for the infrastructure.
have been tagged with meta-data. Note that there are no Mount tags, signifying that these generate virtual directories.

Data attributes are defined by applications or the environment. However, time is a predefined context type with a special syntax. Time is considered separately since a valid duration requires a numeric range. This is partially due to the inability of our directory path syntax to support operations other than equality; a more powerful syntax could support this directly. However, we decided to treat time as a special case and keep the remaining syntax simple. The format for time is \texttt{yyyy-mm-dd[-hh-mm-ss]} or \texttt{yyyy-mm-dd[-hh-mm-ss][yyyy-mm-dd[-hh-mm-ss]]}. For example, \texttt{2002-10-1:2002-10-5} allows a file to be valid from October 1-5 in the year 2002. Once this time is expired, any mount that contains the range is purged from the system so that users are not required to delete old data and mounts.

Several mechanism are available by which to merge and extract personal mount points, which are explained in Section 4.4. By convention, personal data references are added to the \texttt{/users/<username>} namespace. The system automatically creates a user home directory under the \texttt{/users} node. In this way, an application can always be sure that the storage of a user is located in a consistent place in the namespace. When the user leaves a space, the user’s home directory is automatically deleted from the space, which can restrict access unless the user is physically present.

### 4.1.1 Layout Manager Interface

The Layout Manager contains an extensive interface to query for specific mount points and to add/remove mount points, parts of which are described below in Interface Definition Language (IDL). The interface contains three methods for managing the mount table in the database. When a user enters a space, their mounts may be added to the table, thereby making the file be locally available, by using the following method:

```java
void mergeMounts(in string mounts);
```
Table 4.1: Example of several system (left) and user (right) mount descriptions. Entries which do not contain *Mount* tags are used in the construction of the virtual directory hierarchy.
The *mergeMounts* method takes an XML mount description file as an argument. The Layout Manager parses the XML file into mount structures and adds them to the internal table. To obtain a copy of the mounts that belong to a specific user,

```c
string copyMounts(in string owner);
```

can be used, which returns a string containing the mounts in XML format. To retrieve these mounts and additionally remove them from the mount table, the method

```c
string extractMounts(in string owner);
```

can be used. This method is used when a user leaves a space or explicitly wants to remove their mount points from a space to make their data inaccessible, resulting in the allocated user directory disappearing from the system.

The query interface includes various methods to search for mount points. For example, there exists methods to find mounts that match a given sequence of attributes, match the current context, or a certain file index. Each retrieve either a single mount point entry or sequence of mount point entries. A mount point entry is defined by the following structure:

```c
struct MountEntry {
    string owner;
    string mount;
    string host;
    string path;
    long index;
    CtxtSeq ctxt;
};
```

where *CtxtSeq* represents a sequence of context structures, defined as
struct Cntxt {
    string type;
    string value;
};

In file mode, the mounts that do not contain meta-data attributes are queried so that file mode path resolution can be performed by the Resolver. These are returned from a call to

MountEntrySeq getMounts();

where MountEntrySeq is a sequence of MountEntry structures. Two other methods are used to retrieve mounts during context mode operation, which include:

MountEntrySeq getMountsByContext(in CtxtSeq ctxt);
MountEntrySeq getMountsByCurrent(in CtxtSeq ctxt);

The method getMountsByContext retrieves a set of mounts that match a given context. By using getMountsByCurrent, the system can find the mount points that pertain to the current context. Multiple mounts may be returned if files that reside on different machines are tagged with the same attributes. Each of these methods use the algorithm described in Section 3.4.1. For example, if an application were to open the directory /type:/ppt/current:, the getMountsByCurrent method would be called with the argument

Cntxt ctxt;
ctxt.length(1);
ctxt.type = "type’’;
ctxt.value = ‘‘ppt’’;

Several other methods exist in the interface that are used when constructing the virtual directory hierarchy.
4.2 Containers

Data sources are abstracted as containers as exhibited in the C++ Standard Library [SL94, MS96]. Containers represent a data source as a collection of structured data objects and provides data manipulation operations, parsing mechanisms, and content transformations. Each container is a self-contained module and is built as a dynamic link library (on Windows) or a shared object (on UNIX). Containers are loaded on-demand as required by applications, thereby decoupling the containers from system, allowing new container types to be added to the running system without interrupting current applications. A container type defines what kind of objects it contains and how the contents are managed.

Complex processing operations are done on a just-in-time basis, in contrast to just-in-case [PW97]. This lazy processing approach is taken to reduce the potential exponential growth of required processing, increasing latency only on first access or when data content changes. Containers are stateless and may be shared. All per-client information, such as attributes, are send with data access operations.

The file system consists of three container types; file, converter, and context. File containers enable access to native operating system files and directories and parse data of different file formats into indexed data objects. A file container is associated with a data type and encapsulates the logic to deal with that type of data. Untyped containers are available to applications that want to bypass the type system, be it for backward compatibility or due to the lack of an appropriate container type.

Weak devices may not be able to render data in its original format and may require format conversions [Wir]. Conversion of content is performed via converter containers, which are used to transcode data to a new format. Converter containers are instantiated on demand when it is determined that the original data format is not compatible with the type requested by the application and are the basis for the system’s dynamic type mechanism. For example, a small handheld device may want to view a PowerPoint presentation. However, since many
handhelds don't support native viewers and the files can be quite large, the PowerPoint file
could be opened as a collection of GIF images. The GIF images can then be retrieved as
needed, obviating the need for the entire file to be transferred at startup. Complex conversions
may require the support of several converter containers by chaining converters together. The
mechanism for constructing container chains is described in Section 4.3.2.

Data residing in context directories require added processing to resolve the virtual path
and to possibly aggregate various sources, a process performed by proxies called context
containers (a type of file container). The context directory container queries the Layout
Manager for context mount points based on the attributes specified in the directory path.
The results of the query are cached in the proxy and are used to contact remote file servers
running on hosts referenced in the mount points. Inaccessible servers are skipped if they
cannot be contacted within a short amount of time (~5 sec). The context directory container
performs the same role as the Resolver for file mode directory paths. However, since the
operation is considerably more complex, this module encapsulates this functionality and
separates it from the rest of the system. The details of how these proxies manipulate the
virtual directory hierarchy are described in Section 4.5.

4.2.1 Container Structure

The definition of a container interface depends on the type of data objects it holds. The
implementation of a container depends on the data source and therefore different containers
can export the same interface, but have different implementations behind the interface. For
example, a container that wraps PowerPoint presentations, but exports the presentation as
a collection of GIF images, will export a GIFContainer interface, by implementing the mech-
anism to perform the conversion from native PowerPoint format to GIF objects. Figure 4.1
shows the structure of such a container. The PowerPointContainer inherits from both the
GIFContainer interface and the base Container implementation.
Figure 4.1: The structure of containers is composed of interfaces that define data object type and concrete implementations. The UML diagram shows the class hierarchy of the `PowerPointContainer` implementation.

### 4.2.2 Container Interface

The base `Container` interface defines the methods and data structures that are used by all derived container types. These include the structures that define attributes and results of data read and write operations. The `otype()` method defines the output type of a container and is used during container type adaptation. By using the output type of an already instantiated container, a suitable path for conversion can be found to the desired type an application requests. Figure 4.2 shows the IDL for the base container from which all other containers are derived.

Derived containers define the data type and access methods to read and write data objects. Containers must use the name `getObjects` and `putObjects` so that the compiler can generate the correct code from the client-side template classes (described in Chapter 6). Figure 4.3 shows the interface for a `GIFContainer`, which holds GIF images. Note that the attributes are sent with each request for data, allowing container implementations to remain stateless. Such attributes can change the characteristics of the data in some way for fine-grained adjustments and what attributes are supported is defined by the container.
interface Container : UOB::CORBAComponent {

/**
 * The result when getting data from a container
 *
 * count number of items successfully retrieved
 * position position in the container after the get
 * eof 0 = more items remaining; 1 = no more items left
 */
struct GetResult {
    long count;
    long position;
    boolean eof;
};

/**
 * The result when putting data to a container.
 *
 * count number of items successfully sent
 * position position in the container after the put
 */
struct PutResult {
    long count;
    long position;
};

/**
 * The description of an attribute on a container.
 * Attributes may change the way in which data is
 * retrieved or sent. Containers should adjust data
 * based on the attributes and converters should
 * forward them to the next hops in a container chain.
 *
 * name name of the attribute
 * value value of the attribute
 */
struct Attr {
    string name;
    string value;
};

Figure 4.2: The base container interface defines the generic methods and structures used by all derived container types.
typedef sequence<Attr> Attrs;

/**
 * A string representing this container data output type.
 * @return the type
 */
string otype();

/**
 * Tell the size of this container. The size is the number
 * of items in the container. For byte containers, this
 * reports the number of bytes.
 * @return the number of objects in this container;
 * (-1 specifies unknown length)
 */
long size();

/**
 * Close a container.
 */
void close();

Figure 4.2: Con’t

implementation. Therefore, an interface for any new container type must define

- the data object type,

- a sequence of those objects,

- the getObjects and/or putObjects methods that read/write an object sequence.
interface GIFContainer : Container {

/**
 * Description of GIF objects.
 */
typedef sequence<octet> GIFObject;
typedef sequence<GIFObject> GIFObjects;

/**
 * Gets some GIF images from the container.
 *
 * @param buf       the buffer to fill in with data
 * @param position  the position in the source to start
 * @param count     the number of items to read
 * @param attributes attributes affecting data
 *
 * @return           the results of the get operation
 *
 * @throws IOError    an io error occurred
 */
getResult getObjects(out GIFObjects objects, in unsigned long position, in unsigned long count, in Attrs attributes)
    raises(IOError);
};

Figure 4.3: Derived containers define the data object type that a container holds. Read-only containers, such as the GIFContainer, only implement the getObjects method.

4.3 Container Manager

Access to each data source is initiated via a Container Manager that implements the file server functionality described above. The Container Manager acts as a factory for container creation and is the main entry point to gaining access to container object references. Once a manager has successfully created an association between a data source and a file, converter, or context container, a reference to the container is returned from which the contents of the container can be accessed. The Container Manager maintains a list of opened files and
reuses the container if there are multiple requests to use the same data source. This limits
the number of containers that must be instantiated and open native files.

Container Managers also assist in data content adaptation/conversion, as described
above, by finding appropriate converter containers. If more than one converter is required
to convert a source to a desired type, multiple converters are chained together to provide a for-
mat conversion path. Conversion is performed automatically by the manager when a request
to open a container type does not match the underlying data source type. Figure 4.4 shows
several examples that illustrate how containers are connected and accessed by applications.

Figure 4.4: Servers manage their local native files and devices. The system can instantiate
containers on any node in the system and adapt content for different device types. Rounded
rectangles represent container instances. Hexagons represent template wrappers.
4.3.1 Container Descriptions

In order for containers to be linked together to provide the proper conversion chain, a description of the containers must be available. Containers are described using XML; an example is shown in Figure 4.5. Each description specifies the name of the container component (i.e., the name of the library that must be loaded that contains the component), type of the container (file or converter), output data object type, and input data object type or an optional file type (expressed as a file extension) that the container is associated with. When a Container Manager first starts up (or when a new container type is added to the system), it reads the XML descriptions and creates a graph based on the input/output types. This graph is used to determine which containers need to be instantiated and in what order to perform a particular conversion.

```xml
<CFS:Container>
  <CFS:File/>
  <CFS:Interface>GIFContainer</CFS:Interface>
  <CFS:Name>PowerPointContainer</CFS:Name>
  <CFS:Extension>.ppt</CFS:Extension>
  <CFS:Output>gif</CFS:Output>
</CFS:Container>

<CFS:Container>
  <CFS:Converter/>
  <CFS:Interface>PixelContainer</CFS:Interface>
  <CFS:Name>GIF2PixelConverter</CFS:Name>
  <CFS:Input>gif</CFS:Input>
  <CFS:Output>pixel</CFS:Output>
</CFS:Container>
```

Figure 4.5: Example container descriptions for a file and converter container.

A description specifies which container is responsible for a particular data type. The logic to handle that type is encapsulated within the container implementation. Therefore, descriptions specify what container should handle a data type, and the containers specify
how that data is handled.

4.3.2 Container Chains

Using the container descriptions described above, containers can be chained together to provide multiple levels of content conversion. The work presented here is based on our earlier work on content transformation for handheld devices [HRCM00]. The system maintains an in-memory data structure of the current container descriptions to determine if a sequence of containers/converters may be linked together to provide a desired output type for a given data source. The data structure generated is shown in Figure 4.6. The structure consists of an interface map and a graph of data types, where edges between nodes identify the name of a module (i.e., converter container) that performs the transformation. All other nodes represent data types with the exception of the nodes on the extreme right side of the graph which are file extensions.

When a request arrives to open a data source using a particular interface, the desired output type is first determined from the interface map. Next, the graph is traversed to determine if there is a path to the input type (using the otype method) or data source (based on the file extension). Each converter container is instantiated and given a handle to the next container in the chain (see Fig. 4.7) and an object reference to the first container in the chain is then given to the calling process.

Containers may also be converted to a new type after they have already been opened, as shown in Fig. 4.8, by using the adaptInterface method.

4.3.3 Container Manager Interface

The Container Manager is responsible for creating container instances. To create a new container of a given type to access a data source, the method
Figure 4.6: The system constructs a map/graph data structure that is used to determine how to convert content to a desired type.

Figure 4.7: Container Managers instantiate and connect container chains.
Container openContainer(in string name, in string kind,
    in unsigned long mode,
    in unsigned long index);

is used to return a Container of the given kind. The name represents the native file name, which must be obtained previously by resolving the path into its native form. The mode is used to specify how the data source should be opened:

- **Exist** Open a data source only if it exists.
- **New** Create a new data source; fail if it exists.
- **Write** Open a data source for writing.
- **Read** Open a data source for reading.
- **Append** Open a data source for writing and point to end of source.

To adapt an existing container to a new type,

Container adaptInterface(in Container cont, in string kind);
is available. An existing *Container* is passed and a check of the type conversion graph is examined to determine if the *Container* output type can by transformed into the specified kind of interface. If a path is available, a new *Container* is instantiated and connected to the passed *Container*. The new *Container* is then returned to the caller. The Container Manager interface also includes methods to create, remove, and link files and directories:

```c
void mkdir(in string name);
void rename(in string src, in string dst);
void link(in string target, in string link);
```

### 4.4 User Manager

We have implemented several methods by which a user may merge their personal data into the local space. One method allows users to carry their own personal mounts with them via a handheld (see Fig. 4.10(a)). We have developed an application for PocketPC devices that is used as the conduit for transporting mount points, as shown in Fig. 4.9. When a user enters a space, the device obtains a handle to the space via IR beacon which is the entry point to all services running in the space and is used for further communication with the infrastructure via the 802.11 wireless network. The application includes graphical controls to merge mount points residing on the handheld into the space file system and to extract mount points from the space and store them on the handheld.

An alternative methods does not require users to carry any handheld device with them (see Fig. 4.10(b)). Users are authenticated in the system via finger print detector through an authentication service (AS). The mount server receives the event when a new user has entered the local space and the user mount points are retrieved from a profile server (PS). Both the AS and PS are domain service that support multiple spaces.

The profile server is implemented as a component called the *User Manager* and is used
to support user mobility. The User Manager stores information about the host and path to the file that contains the user’s personal mount points. The Layout Manager consults the User Manager, given a user name, and retrieves the host/path information and then contacts the Container Manager running on the host to retrieve the mount point descriptions. Once retrieved, the descriptions are merged with the descriptions in the mount table, thereby making the data available to local applications. User information is described in XML format, as shown in Fig. 4.11. The Path tag specifies the XML file that contains the user mount point descriptions.

Depending on the method used, the user can control how automated the process is. In the first method, the user must manually merge the data mount points by explicitly pressing a button, while in the second case, the merge operation is performed automatically. By allowing the user to make the decision, we allow them to determine how much control to retain.
Figure 4.10: The architecture of CFS consists of the mount server (MS) and file server (FS). The mount server defines the layout of storage for a space via mount entries, which contain pointers to storage on remote file servers. A mobile handheld can be used to carry personal mount points, which may be merged into the local namespace. When users enter without any device, the system can use the authentication service (AS) and profile service (PS) to retrieve user mounts from a home server.

### 4.5 Virtual File Hierarchy

The system uses the meta-data tags attached to files to construct the context mode virtual directory structure and handles paths differently depending on whether the last component of the path is a attribute type or value.

```xml
<CFS:Profile>
  <CFS:Owner>ckhess</CFS:Owner>
  <CFS:Host>smokey.cs.uiuc.edu</CFS:Host>
  <CFS:Path>C:\Users\ckhess\CFSLayout.ckhess.xml</CFS:Path>
</CFS:Profile>
```

Figure 4.11: An example user profile that contains the location of user mount points.
4.5.1 Context Files

If the last component is a file name, a ContextByteContainer is instantiated. ContextByteContainers are proxies for native files that reside in a context directory (i.e., files with attribute meta-data attached to them). The proxy resolves the context path by querying the Layout Manager to find the mount that contains the attribute tuples in the path. This mount contains the host machine and native directory which is used to find the native file. Context files are of the form [/<type:>/<value>]/<filename>, for example, /location:/2401/gaia.ppt.

4.5.2 Context Directories

Context directories represent the virtual hierarchy constructed from the context meta-data tags attached to files. The ContextDirectoryContainer implements a context directory and is the most complex of all the containers. When retrieving directory entries from a context directory, the container runs different algorithms based on the structure of the path. There are five context directory types that are derived from the structure of the path and are summarized below.

- **Context Type** A context type directory is any in which the path ends with a component of the form <type:>. A directory of this form contains all context values that have the given type (and all others in the path) as part of their meta-data description. For example, the directory /location: contains all context values that are part of the location/<value> tuple.

- **Context Value** A context value directory is one in which the path ends with a component of the form <value>. This directory contains all files with the given (subset of) attributes attached to them, in addition to all attributes that are a part of mounts that contain the tuples that appear in the path. Two more directories are also available, specifically the current:, which represents data in the current context, and users:
directories, which represents the context files for one user. For example, the directory `/location:/2401` contains all files with this specific tuple attached, as well as any other types that contain the `<location/2401>` tuple in the meta-data tag. For example, if there existed a meta-data tag of `<location/2401>` and `<situation/seminar>`, the `<location/2401>` directory would also contain the directory entry of `situation:`. This allows navigation in all combinations of tuples.

- **Root** This is the root directory in context mode. This directory returns the same entries as a context value directory, minus the `current:` and `user:` directories.

- **Current** This directory contains files associated to the properties in the path and any active environmental context. An example is `/location:/2401/current:`.

- **Data** Directories can be tagged with meta-data, thereby allowing a convenient mechanism by which a collection of relevant files can be grouped together. These directories can be entered, and navigated further if there are sub-directories within. Any of these sub-directories are considered data directories. They are a mixture of context and file modes; the attribute portion of the path directs the navigation to a specific file server and the remaining path components specify specific data storage on that file server. An example of this type of directory is `/location:/2401/music/rock`.

### 4.5.3 Directory Listing

When the last part of a path is an context type (e.g. location), the system queries the mount server for all mounts that contain tags for the given context. The directory entries returned by the module are those contexts that appear in the context type tag (only a single entry for redundant entries are returned by the mount server, i.e., when different users have files with the same context attached to them). For example, when an application opens the `/location:` directory, the mount server returns all mounts that include a location tag and the directory entries are the result of the values associated to the tag; the contents of the
directory includes all location values that are attached to files. The following listing signifies that there are files in the system that have one (or more) of the four different room context locations attached to them:

```plaintext
> cd /location:
> ls
2401 2402 1310
1320
>
```

Listing a context value directory (e.g., 2401 in the above example) shows what files actually have the given context attached to them, as specified by the path. The module queries the mount server for all mounts that match the given context and then contacts all file servers specified in the mounts to retrieve the names of all files and aggregates them to make it appear as if they are located together. In addition, any remaining context names that are not in the current path are also returned, so that they may be navigated further. For example:

```plaintext
> cd /location:/2401
> ls
situation: time: group:
space: current: users:
cfs.ppt demo.mpg
>
```

From this example, `cfs.ppt` and `demo.mpg` have the context of `location == 2401` attached to them, which may be located on different file servers. Since there is no fixed order for navigating the context hierarchy, opening `/space:/office/group:/srg` is equivalent to `/group:/srg/space:/office.`
4.5.4 Directory Creation

Next we address how context directories are created. When an application creates a virtual directory, the system creates a mount point that contains context type/value tags as specified in the path. This mount acts as a place holder until the context is eventually attached to files (described in the following section), so that the system can display the context directory if no files are yet added. For example, creating a directory /location:/2401/situation:/meeting creates the mount shown in Fig. 4.12.

```
<CFS:Storage>
  <CFS:Owner>ckhess</CFS:Owner>
  <CFS:Context>
    <CFS:Type>situation</CFS:Type>
    <CFS:Value>meeting</CFS:Value>
  </CFS:Context>
  <CFS:Context>
    <CFS:Type>location</CFS:Type>
    <CFS:Value>2401</CFS:Value>
  </CFS:Context>
</CFS:Storage>
```

Figure 4.12: An example mount point generated when creating a virtual directory with location and situation context.

This mount then gets added to the mount server, which is specific to the user that created it and becomes a part of their set of personal mount points. Deleting the directory removes the mount from the mount server.

4.5.5 Attaching Context

The operation of explicitly attaching context to files is handled by the copy operation, which is a primitive available in the CFS interface. Copying a file to a context directory attaches the context associated with the path to the file by creating a directory on disk for that context and creating a link to the real file in the generated directory. A mount is then generated
with the context from the destination directory path, information about the host machine containing the newly created directory, and the path to the directory. If a mount is already available for the given context, the system will simply reuse the associated directory and add the link to it. The data and links to data reside on the same machine, regardless of the space in which the attachment is initiated. As shown in Fig. 4.13, the generated directory \( C:\Temp\15687 \) is used for all files (and links to files) that have the context of meeting and 2401 attached to them (on the given host).

   <CFS:Storage>
   <CFS:Owner>ckhess</CFS:Owner>
   <CFS:Host>srg181</CFS:Host>
   <CFS:Path>C:\Temp\15687</CFS:Path>
   <CFS:Context>
      <CFS:Type>situation</CFS:Type>
      <CFS:Value>meeting</CFS:Value>
   </CFS:Context>
   <CFS:Context>
      <CFS:Type>location</CFS:Type>
      <CFS:Value>2401</CFS:Value>
   </CFS:Context>
   </CFS:Storage>

Figure 4.13: An example mount point generated when attaching location and situation context to a file.

We use mounts to store context information rather than directories on disk because context directories are not hierarchical and having the information in the mount points makes finding and aggregating files with a particular context easier and more efficient.

Opening the \( /situation:/meeting/2401 \) directory results in the file system opening the directory \( C:\Temp\15687 \), which may contain files or links to files, following any links, and displaying the files and pointed to files (the same can be done with directories)\(^2\).

Figure 4.14 shows the result of the \( /current: \) directory when the context of the space is \texttt{situation == meeting \&\& location == 2401} and how files are mapped into the direc-

\(^2\)Directories located within the virtual hierarchy may be opened to locate real files in sub-directories.
Figure 4.14: Context directories aggregate all files with the given context tags. Files may be owned by different users and may reside on remote file servers. Context mount points reference directories that contain files and links to data with the associated context.

tory using the mount points described in Table 4.1. The native directory $C:\Temp\15687$ is a generated temporary directory that was created when a file was copied to the $\text{/situation/meeting/location:/2401}$ context directory.

4.6 Prototype Environment

We have implemented our system on Windows 2000, with the mount and file servers developed as application-layer servers written in C++. File servers access local data through the native file system and export portions of the storage to the space file system. CFS is a part of Gaia [RC00, RHC\textsuperscript{+}02], a ubiquitous computing middleware infrastructure we have devel-
oped that turns physical spaces and the resources they contain into a single programmable system. *Gaia* is similar to traditional operating systems by managing the tasks common to all applications; more details regarding these services is given in the next chapter. Applications are built in a generic way that are able to run in arbitrary active spaces by mapping them onto the available resources [HRC02]. We have deployed *Gaia* in a prototype room containing large four wall-mounted plasma displays, 5.1 audio system, 15 Pentium-4 PCs, video wall, IR beacons, badge detectors, wireless and wired Ethernet network, and X-10 devices.
Chapter 5

Gaia Operating System

The context file system is part of Gaia, a meta-level operating system to support the development of applications in ubiquitous computing environments. Gaia is composed of a collection of kernel services and provides well-defined interfaces to these services so that applications may be built in a generic way that are able to run in arbitrary active spaces that run the infrastructure. The kernel service core includes execution nodes, events, entity presence (devices, users, and services), authentication, context notification, location, discovery, trading, and naming, as shown in Fig. 5.1. These services are started through a distributed bootstrap protocol that initializes the Gaia infrastructure.

Figure 5.1: Gaia provides core services for building ubiquitous applications.
In this chapter, we give an overview of the *Gaia* architecture, a brief description of the kernel services, and the protocol used to bootstrap the system.

### 5.1 Component Management Core

The lowest layer of *Gaia* is composed of a core that consists of a substrate that manages component lifecycles, instantiation, and destruction, called the *Component Management Core* (also known as the UOBHost for historical reasons). This core acts as a remote execution node that can be instructed to load and run components and forms the basis of *Gaia*'s distributed applications and services. Applications are typically composed of components running on distributed nodes and are connected together via an object bus for remote communication. When new components are loaded into the system, they are registered with the system and can be accessed by local and remote software entities.

All *Gaia* components communicate via CORBA remote method invocations. CORBA is recognized to have several known limitations and we therefore have added some functionality to better support ubiquitous applications. For example, *Gaia* incorporates a soft-state beaconing mechanism to detect when components have crashed or left the system. Although our current implementation uses CORBA, *Gaia* could be constructed with any distributed object architecture, such as RMI [Sun], .NET [Mic], or DCOM [Tha97].

### 5.2 Event Manager

The *Gaia* infrastructure and applications rely heavily on events. Events are used to decouple components within the system by communicating via asynchronous messages and are particularly useful when a producer of information does not know what components are interested in that information. They also are helpful in distributed environments where components are dynamic and come and go on a regular basis, either intentionally or as a result of system
failures. *Gaia* implements an event distribution mechanism that is based on the CORBA Event Service [Thea] called the *Event Manager* 5.2. This service is based on the concept of event channels, which are distribution channels that are named in a two-layer hierarchy. The first level of the hierarchy is used to categorize a collection of channels, such as error or application specific channels. Channels can be further separated to provide fine-grained functions. For example, error channels can be split into warning, alert, and critical errors. *Gaia* defines several standard channels, which include discovery, presence, error, component creation, component destruction, application change, context, and filesystem change.

![Event Manager diagram](attachment:image.png)

**Figure 5.2:** The Event Manager facilitates the manipulation of event channels.

The Event Manager provides some utility functions that facilitates the creation and destruction of channels, as well as sending and receiving channel events. All events maintain a common header, with specific channels able to define their own payload type, making them very flexible. In addition, persistent events are supported, thereby allowing a component to search for past events. For example, an application may wish to know when a certain person entered a room.

## 5.3 Presence Service

Active spaces are populated by many entities, including people, software components, and hardware devices. Some of these entities are permanent and available for use as long as the system is running, while others may be transient and mobile. Monitoring of these entities is
required to recognize their current state, when they are active, and if they are present in a physical space.

Software components are monitored through a heartbeat mechanism that are processed by the Presence Service. When the heartbeats of an entity are received, the arrival of the entity is announced through the use of event channels. Three presence channels are defined for software and hardware components; application, device, and service channels. When an entity’s heartbeats are not received for some time, it is assumed to have stopped and its departure is announced on the appropriate channel. These enter and leave events signal other components in the system of the presence of various entities in the system.

![Diagram](image)

Figure 5.3: The Presence Service listens for the availability of physical and digital entities.

Different sensors, such as fingerprint detectors, USB devices, and smart watches, are used to detect the presence of people. Once users are detected, an event is sent on the person presence channel. The Person Tracker processes these events, which includes keeping track of the current location of different people, trying to refine the location information by combining various reports, and trying to infer the identity of persons. It then posts its findings on the person discovery channel.
5.4 Space Repository

The *Space Repository* is a centralized database containing information about all active devices and services in an active space (see Fig. 5.4). It keeps this information up-to-date by listening on the presence channels described above, on which events about new entities, as well as entities that are no longer active, are sent. All entities in the system have an associated XML description, which includes properties such as entity type, name, location, etc. The Space Repository can be queried for entities based on these properties. For example, an application may require two large displays, which can be found by querying the Space Repository for two display that satisfy a particular dimension constraint. Note that in different spaces, different displays may result from such a query depending on what resources are available. Therefore, applications can find local resources so that they may be mapped onto the current space.

![Diagram](image)

Figure 5.4: The Space Repository stores information about all active entities in an active space.
5.5  Context Service

Applications make use of contextual information to affect their behavior and adapt to environmental conditions. The *Gaia Context Service* allows applications to subscribe to or query for context information. For example, an application may wish to be notified that a meeting has begun so that the agenda may be placed on the wall displays. Context is represented using a causal model based on the 4-tuple *context type, subject, relater, object*. The context type is a category of context and is mapped to an event channel. The remaining three items describe how the context is defined. For example, a valid context could be *location, jane, entering, RM 2401*. The structured nature of context descriptions allows first order logic (e.g., unification) to be applied to sets of context information. This allows more complex context relations to be constructed from simple descriptions.

5.6  Security Services

The complexity of securing a ubiquitous environment is increased due to dynamism and context and is vital to the acceptance of ubiquitous computing systems. *Gaia* provides support for authentication and access control. Authentication in *Gaia* is achieved through a number of different methods, including biometrics, small personal devices, and face recognition. Access control is currently under development and involves the use of role-based access control (RBAC) to define permissions for users of the system. Depending on the context of a given situation, the role of a user may change and therefore alter their security credentials.

5.7  Application Framework

Traditional desktop applications are typically constructed as monolithic entities. However, in an active space environment, users move, there are a variety of output devices, and a choice of input modes; traditional application construction techniques are inadequate. *Gaia*
applications are constructed from a distributed application framework that is tailored to the unique requirements of ubiquitous applications. The Gaia application framework is inspired from the Model-View-Controller pattern [HH95] often used in the construction of graphical applications. The Gaia framework generalizes the view into a presentation, that is, a presentation is output that can be sensed, i.e., through sight, hearing, touch, etc. In addition to these three components, the framework adds a fourth component called the coordinator. The coordinator is responsible for maintaining the application bindings and can dynamically remove and/or add new bindings after the application has started. For example, presentations can be added to a slide show from a handheld device after the show has begun.

Applications are described by generic requirements, such as what components are needed, how many of each are allowed, what types of special hardware is required (e.g., speakers), etc. These descriptions are then “specialized” to the resources available in a particular space [RC02]. This process is accomplished by querying the local repository to determine what resources are present and what their capabilities are. Users can generate application configurations that are customized to a certain space. For example, a configuration could instantiate a slide show presentation to start on all available displays with a controller started on an available handheld device.

Application configurations specify what and where components should be instantiated. Once the components are running, they are glued together so that they may communicate. The instantiation of application components is achieved through use of the Lua scripting language [IFC96, IFC99]. A binding between Lua and CORBA was developed, called LuaORB [CRI97], that provides support for communication with distributed CORBA objects from Lua scripts. Lua scripts allow applications to be composed of instructions that specify on which remote execution nodes application components should be launched and describe how to tie those components together. Lua provides a convenient mechanism to more easily construct dynamic distributed applications.
5.8 Bootstrap

Before applications may execute in an active space, the machines hosting the infrastructure must be running and the core kernel services must be started. The *Gaia Bootstrap Protocol* (GBP) is used to automate the steps needed to start the kernel services required by *Gaia* applications. An active space creates a geographic area network (GAN) on top of the underlying local area network (LAN) where each active space is able to function autonomously; that is, an active space need not depend on any outside services to be available for correct operation of the local space. For example, an active space can run on a computer that is totally disconnected from the network. In addition, a physical space, such as an office, may be split into several active spaces. Even if they share the same underlying network, they are able to operate independently from one another. A problem that GBP addresses is how to limit an active space to a portion of the underlying network, i.e., the reverse scaling problem [KF02].

The *Gaia* kernel is composed of several essential services. The order in which they are started is important, since there are certain dependencies among them. Before a service can be used, it must be discovered and a reference to it must be obtained through a directory service. The directory service used in *Gaia* is the CORBA Naming Service (NS). The NS acts as a central repository where kernel services can be registered and discovered. Services are registered within a space with well-known names under the root naming context, allowing them to be discovered in an identical manner in each space. Once an object reference is obtained, communication continues directly between the client and the service component. Although the NS may appear to be a bottleneck in the system, this is typically not the case, since in *Gaia* there exists one NS per space (rather than having one NS per LAN), and the NS is only contacted to retrieve or register an object reference.

Services are categorized into two types; *space services* and *host services*. There exists a unique instance of each space service per active space; host services run on a specific machine
in a space and there may be more than one instance of such a service per active space. For example, a device or sensor may be attached to a particular machine and the service that exports it must be run on the same machine to which it is connected. The following table enumerates the space and host services that are started as part of the bootstrap process, including the order and dependencies.

<table>
<thead>
<tr>
<th>Order</th>
<th>Service</th>
<th>Dependency</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Interface Repository</td>
<td>None</td>
<td>Space</td>
</tr>
<tr>
<td>2</td>
<td>UOBHost</td>
<td>Interface Repository</td>
<td>Host</td>
</tr>
<tr>
<td>3</td>
<td>Naming Service</td>
<td>UOBHost</td>
<td>Space</td>
</tr>
<tr>
<td>4</td>
<td>Event Manager</td>
<td>Naming Service</td>
<td>Space</td>
</tr>
<tr>
<td>5</td>
<td>Presence Service</td>
<td>Event Manager</td>
<td>Space</td>
</tr>
<tr>
<td>6</td>
<td>Trading Service</td>
<td>Naming Service</td>
<td>Space</td>
</tr>
<tr>
<td>7</td>
<td>Space Repository</td>
<td>Trading Service</td>
<td>Space</td>
</tr>
<tr>
<td>8</td>
<td>Layout Manager</td>
<td>Naming Service</td>
<td>Space</td>
</tr>
<tr>
<td>9</td>
<td>Container Manager</td>
<td>Layout Manager</td>
<td>Host</td>
</tr>
</tbody>
</table>

Table 5.1: Kernel services that are started as part of the bootstrap process must be started in a particular order to ensure that the dependencies among services is preserved.

For proper operation of the kernel, there must be a unique instance of each space service running. GBP ensures this property by probing for existing instances of a service before starting them. A space can be configured depending on the available machines and the desired layout of the machines in the space. For example, all services can run on one machine, or may be distributed on different machines within the space. The bootstrap process does not complete until all services are running or an error is detected. GBP uses the Lua scripting language to start each of the services. The bootstrap scripts are split into five parts:

- **boot.lua** - the main driver script to run the protocol.

- **bootstrap.lua** - the functions to perform the bootstrap process.

- **services.lua** - list and configuration of services to be started.
• *as.cfg.lua* - the space specific configuration.

• *local.cfg.lua* - machine specific local configurations.

Each space is assigned a unique space id (SID), which can be a room number in a building, airport code in a flight terminal, etc. The *as.cfg.lua* configuration script contains the SID and *services.lua* contains a list of the services that are to be started and any special configuration information that is required for each. The configuration for each service specifies the machines that the service is allowed to run. This list is used to determine if the current machine should attempt to start a service if it is not already running in the system.

The bootstrap process iterates over the list of services that are to be started. The protocol first probes the network to see if the service is already running. Depending on the service, this is done in different ways, and is described below. If the service is determined to be running already, the protocol moves to the next service in the list. If the probe of the service is unsuccessful, the protocol checks to see whether the current machine has been designated to run the service. If so, the service is started on the local machine, otherwise the protocol waits for the service to be started by another machine in the network.

### 5.8.1 Interface Repository

The first service that must be started is the CORBA Interface Repository, which is required by LuaORB so that it can perform type checking of the service interfaces that it needs to communicate with. Since the Interface Repository is started before the NS, it cannot be registered in the NS and therefore is started on a known host and port.

### 5.8.2 UOBHost

After the Interface Repository is started, an execution node (i.e., UOBHost) is started on the local machine and is used to start all subsequent services. The UOBHost is used to start local and remote components and containers for components. Since the UOBHost is
started before the NS, it must have no dependency on the NS and is therefore also bound to a well-known port on the local machine. In order to start a component on a particular (possibly local) machine, a CORBA request is sent using the iioploc mechanism, therefore bypassing the need for the NS to contact a UOBHost. This does not violate the location transparency exhibited in the other services since the machine name must be known in order to start a component on it.

5.8.3 Name Service

Since the NS is the central database for storing object reference, it must be started before references can be entered into it. Discovering CORBA name services is typically achieved through multicast, where the service listens on a well know multicast group address and port. This approach does not work in the AS environment, as explained above, since there may be multiple GANs (and hence multiple NSs) per LAN. In order to avoid collisions, each NS must respond only to queries for the space that it is supporting.

Gaia was designed to use off-the-shelf services. For example, Gaia uses standard implementations of the CORBA naming, trading, and event services. However, since many implementations use multicast for discovery, they are restricted to one service per LAN. In order to circumvent this limitation, Gaia employs a Name Agent (NA) that is used to filter requests and only responds to those requests originating within the space it is supporting. When the NS is first started, it writes its object reference to a file. The NA is configured to respond only to requests for a given SID, which is placed in each request. The NA is run on the same machine as the NS and waits for this file to become available and then reads the object reference into memory. The NA then listens on a well-known multicast port and address, receives all requests, and filters them based on the SID it is configured with, and responds with the reference to the NS if the SID in the request matches its own.

The steps taken to resolve the NS are shown in Fig. 5.5 and operate as follows:
Figure 5.5: Each space is identified with a unique space id to bind a collection of services belonging to the same space.

- The NS is started via the UOBHost in a space if no other instance is determined to be running. The object reference of the root naming context is written to persistent storage.

- The NA is started and listens on a different port than the NS. The NS port is not used, but is made different to eliminate port allocation collisions. The NA reads the object reference from persistent storage. The NA is configured to filter out requests not directed for it. The NA will periodically look for the existence of the object reference. If it is not available, perhaps because the NS has not started yet, it will periodically poll until it is found. Once the object reference is found, it is cached in memory and the NA continues by joining a multicast group and binding to a certain port.

- A component sends out a request to discover the NS (through the use of a client library). It waits for a limited amount of time for a response.

- The multicast request is send on the LAN. All NAs tuned in to the multicast group receive the request for the NS object reference.

- Each NA filters the request based on the SID that is encapsulated within the request. The NA that has been configured with the same SID accepts the request.
• The object reference of the NS is unicast back to the requester in a UDP packet. If the requester does not receive the response within a specific timeout period, it attempts to resolve the NS two more times. A response may not arrive either because a packet was lost or there is no NA for the space. If, after three attempts to contact the NA and no response is returned, the NA/NS is assumed not to be running. If a response is received, the NS object reference is cached and probing is thereafter bypassed.

5.8.4 Other Services

All remaining services are registered in the NS. The probing mechanism used to discover these services simply queries the NS to check if they are already registered in the NS. When a service registers itself, it discovers the NS by using the mechanism described above. Therefore, if a service reference is found in the NS, it is assumed to be running in the space. The potential for race conditions is eliminated through the object binding mechanism; if an object tries to bind itself and an object of the same type is already bound, the binding will fail. Therefore, the first received bind request will succeed; failure to bind will signify that another process has bound itself in between the time that the probe failed and the bind was issued.

5.8.5 Mobile Devices

Mobile devices are treated differently from stationary machines, since they typically communicate through wireless links, such as IEEE 802.11 [Pag] or bluetooth [Blu01]. As a result, it may not be possible to multicast from the wireless network to the network underlying the space, due to administrative domains or other technical hurdles. In such situations, the device must use an alternate method to obtain a handle to a space; one method is through the use of infrared beacons. Once the NS has been activated in a space, its object reference is periodically beaconed into the space via infrared. Once a mobile devices picks up the
beacon, it can be used to obtain any of the other services or components that are available in the space. Upon receiving the object reference, the mobile device registers itself in the Space Repository so that it can be used by the other components in the system.
Chapter 6

File System Interface

To conceal the details of the CORBA architecture from the developer, access to Gaia services is facilitated through a set of wrapper classes that comprise the application programming interface (API) to Gaia. In this section, we describe in detail the structure of the wrappers for the file system. These wrappers employ a combination of C++ templates and generic programming [Mus89] concepts that hide the communication infrastructure and maximize code reuse. The API provides a uniform interface for accessing different data types, for manipulating file mode and context mode directories, and for providing transparent format transformations. Two main interfaces exist that 1) allow developers to write applications that access data through the CORBA containers (container wrappers) and 2) allow applications to manipulate the file system (file system wrappers).

6.1 Gaia Container Wrappers

The container wrappers present the container abstraction to the developer on the client side, where there exists various container types for different underlying data types, each of which exports the same base interface. The main way in which containers differ is in the types of data objects that they contain. Therefore, these client containers inherit from a template class that is parameterized on the container data object type. The template maintains a reference to the underlying CORBA container and provides methods for creation,
adaptation, reading, and writing. As shown in Fig. 6.1, the template parameter list consists of the CORBA container type (C) and data object type (O). Creating a container wrapper consists of inheriting from the base container template class and specifying the data object type in the parameter list. Through the use of a macro, this can be accomplished in one line of developer code.

Templates are used to provide compile-time polymorphism of CORBA container types, thereby applying generic programming techniques to distributed objects. CORBA containers provide methods to get and put objects of a particular type. The name of the methods must adhere to a convention (getObjects() / putObjects()) for each container so that the template classes can access them in a uniform manner. Since the particular object types to be transferred are specified in the template parameter list, the template interacts with the underlying CORBA container to transfer data of a certain type when communicating with the
remote object request broker. In effect, the user-level container provides a consistent view of a CORBA container, although objects of different types are specified in the IDL container descriptions and are marshaled over the network. The need to use the CORBA type Any to transfer objects is removed and eliminates the need to typecast objects to a specific type. The complete client container interface can be found in Appendix A.1.

![Diagram of Container Wrappers and Marshall](image)

Figure 6.2: User-level containers are described using generic programming concepts to maximize core reuse. Typed objects are marshaled over the network. Groups of data objects can be cached in the local (generic) container.

A variety of container types have been implemented to determine the issues involved in constructing different containers and to validate the ability of the interface to support different data types. The containers that are currently implemented are listed Table 6.1.

When a container is opened, the interface wrappers deal with resolving the path name by using the mounts obtained from the Layout Manager and contacting the appropriate Container Manager to create an instance of the CORBA container.

### 6.2 Gaia File System Wrappers

The file system wrappers provide an interface for general file system operations, such as changing the current directory, determining the home directory, adding and removing mounts, etc., and are part of the GaiaFS namespace. For example, to change the current working
## Container Catalog

<table>
<thead>
<tr>
<th>Interface</th>
<th>Library Name</th>
<th>Type</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteContainer</td>
<td>ByteContainer</td>
<td>file</td>
<td>*</td>
<td>byte</td>
</tr>
<tr>
<td>DirectoryContainer</td>
<td>DirectoryContainer</td>
<td>file</td>
<td>*</td>
<td>dirent</td>
</tr>
<tr>
<td>ContextByteContainer</td>
<td>ContextByteContainer</td>
<td>file</td>
<td>.bctxt</td>
<td>byte</td>
</tr>
<tr>
<td>ContextDirectoryContainer</td>
<td>ContextDirectoryContainer</td>
<td>file</td>
<td>.dctxt</td>
<td>dirent</td>
</tr>
<tr>
<td>TextContainer</td>
<td>TextContainer</td>
<td>file</td>
<td>.txt</td>
<td>text</td>
</tr>
<tr>
<td>TextContainer</td>
<td>NewsContainer</td>
<td>file</td>
<td>.news</td>
<td>text</td>
</tr>
<tr>
<td>GIFContainer</td>
<td>PowerPointContainer</td>
<td>file</td>
<td>.ppt</td>
<td>gif</td>
</tr>
<tr>
<td>PixelContainer</td>
<td>GIF2PixelContainer</td>
<td>file</td>
<td>.gif</td>
<td>gif</td>
</tr>
<tr>
<td>PixelContainer</td>
<td>GIF2PixelConverter</td>
<td>converter</td>
<td>gif</td>
<td>pixel</td>
</tr>
<tr>
<td>BitmapContainer</td>
<td>Pixel2BitmapConverter</td>
<td>converter</td>
<td>pixel</td>
<td>bitmap</td>
</tr>
<tr>
<td>MPEGContainer</td>
<td>MPEG2PixelConverter</td>
<td>converter</td>
<td>mpeg</td>
<td>pixel</td>
</tr>
<tr>
<td>PageContainer</td>
<td>PageContainer</td>
<td>converter</td>
<td>text</td>
<td>page</td>
</tr>
</tbody>
</table>

Table 6.1: Catalog of implemented containers.

directory, an applications may call \texttt{GaiaFS::chdir( <newdir> )}. Other frequently used methods are

```c
void mount(in MountEntry &entry);
```

which adds a new mount into the mount table and

```c
string localize(in string name, in long index);
```

which makes a local copy of a file, so that it can be used by native applications. The complete interface with a brief description of each method is given in Appendix A.2.

### 6.3 Interface Usage

In this section, we show examples of how the container interface is used. To gain access to a particular data source, an application instantiates a container of the desired type. In the following examples, error handling is removed for clarity.
6.3.1 Attaching Context

Attaching context is accomplished via the `copy` operation, which is a special method available in the `DirectoryContainer` interface. The `copy` operation allows a file mode file to be copied to a context mode directory and internally generates a context-aware mount point and sets up the proper links in the underlying system, as described in Section 4.5.5. The following code fragment illustrates the necessary syntax:

```c
const char *filename = <file mode file name>
const char *dirname = <context mode directory name>
DirectoryContainer d(dirname, GaiaFS::Read);
d.copy(filename);
d.close();
```

6.3.2 Accessing Directories

The following code fragment illustrates the way in which an application can open a directory and read its contents.

```c
int count;
Dirents dirent;
DirectoryContainer c(<filename>, GaiaFS::Read);

GaiaFS::chdir(<dirname>);
while (!c.eof()) {
    count = c.get(dirents, 16);
    for (int i = 0; i < count; i++) {
        cout << dirents[i].name << endl;
        cout << dirents[i].size << endl;
        cout << dirents[i].ctime << endl;
    }
}
```
A `DirectoryContainer` holds objects of type `Dirent`, whose structure is as follows:

```c
struct Dirent {
    long   atime;
    long   ctime;
    long   mtime;
    long   size;
    long   type;
    string name;
    long   index;
};
```

Note that a `DirectoryContainer` can open both file and context mode directories. The system will instantiate the correct module for the application developer based on the path structure. The `atime`, `ctime`, and `mtime` specify the last access time, creation time, and last modification time, respectively. The `size` is the number of data objects that the container holds and the `type` describes the file type, which may be one of the following:

- **Dir** The source is a directory.
- **File** The source is a file.
- **DirLink** The source is a directory link.
- **FileLink** The source is a file link.
- **VirtDir** The source is a virtual directory.
• \textit{CtxDir} The source is a context directory.

The \textit{name} is the name of the file or directory and the \textit{index} is a system defined integer used to avoid name clashes.

\subsection*{6.3.3 Accessing Bytes}

To list the contents of an unstructured file, the \textit{ByteContainer} may be used.

\begin{verbatim}
int count;
Bytes bytes;
ByteContainer c(<filename>, GaiaFS::Read);

while (!c.eof()) {
    count = c.get(bytes, 256);
    cout.write(bytes, count);
}
c.close();
\end{verbatim}

\textit{Bytes} are defined as a sequence of \textit{Byte} objects, where a \textit{Byte} is defined as an \textit{unsigned char}. The implementation of \textit{Bytes} has been customized to obtain better performance since it is used often.

\subsection*{6.3.4 Format Conversion}

Format conversion is performed transparently to the developer. Any data source can be opened with a particular container type to convert the data objects to the type associated with the container (if the conversion is supported). For example, a handheld device may not support a native PowerPoint viewer or may wish to read slides on demand rather than opening the entire file. In this case, the file may be opened with a \textit{GIFContainer} so that the
application only deals with GIF image objects. The following example illustrates how this is accomplished.

GIFs gifs;
GIFViewer viewer;
GIFContainer c("cfs.ppt", GaiaFS::Read);

c.satt("width", "160");
c.satt("height", "240");

while (!c.eof()) {
    c.get(gifs, 1);
    viewer.display(gifs[0]);
    OS::sleep(5000);
}

The satt allows data type attributes to be set, in this case specifying the image dimensions. The next example further illustrates the file system dynamic data types. In this example, a presentation is opened using a bitmap container, which contains bitmap objects. The file server transparently instantiates a sequence of modules that can perform the appropriate conversion to the desired type (i.e., ppt → gif → pixmap → bitmap).

Bitmaps bitmaps;
BitmapViewer viewer;
BitmapContainer c("cfs.ppt", GaiaFS::Read);

c.satt("width", "160");
c.satt("height", "240");
while (!c.eof()) {
    c.get(bitmaps, 1);
    viewer.display(bitmaps[0]);
    OS::sleep(5000);
}

6.4 File System Events

The file system uses events to notify applications of changes in the namespace. These changes take place when users enter and leave a space and their personal data is merged into the namespace, when new files are added, and when files are deleted. Applications register for namespace change events through the \texttt{GaiaFS::addListener} method by passing an object that implements the \texttt{GaiaFS::Listener} interface. This interface contains one method, \texttt{onChange}, which gets called each time the namespace has been altered. This allows applications to immediately update their view of the namespace. For example, a file browser can automatically update the current view of the file system whenever a new user enters a space.
Chapter 7

Applications

A collection of applications have been developed for Gaia, a part of which comprise a suite of applications that are used during our weekly seminars and meetings. During our seminar, we discuss several papers and show them on four large wall-mounted displays. We can select the paper to view and control the synchronized views from handheld devices. When users enter the space, they are authenticated via biometric fingerprint detector and visual feedback is provided by a ticker tape that circulates sequentially across the displays. Authenticated users are also logged so that we can record the attendance of participants. We also provide support for displaying presentations on any number of additional displays or handheld devices. In this chapter, we describe several of these applications and how they leverage the features of the file system, thereby gaining support for user mobility, data organization, and heterogeneous devices.

7.1 Gaia Data Browser

We have implemented a graphical data browser similar to the Windows Explorer used to navigate the file hierarchy, as shown in Fig. 7.1. Our browser operates in both file mode and context mode which may be toggled by using the two buttons in the top left corner, allowing users to easily copy a file to a context directory. In file mode all files users have imported into the space are visible under the allocated user directory. Context mode represents the virtual
file hierarchy constructed from the available context meta-data and is used for creating
new context directories, attaching context to files by copying them to a context directory,
and discovering what context is attached to files. New context directories are generated by
creating a new folder, which internally calls the `mkdir` operation that adds a new mount
point to the mount server, as described in the previous chapter.

![Image of file explorer](image)

**Figure 7.1**: The virtual file hierarchy is constructed from the meta-data tagged to files and
directories. New meta-data can be added by creating new context directories and copying
files into them.

We use a calendar to automatically schedule and launch our seminar applications at
the prescribed time. However, we are also able to manually launch applications from our
browser, as shown in Fig. 7.2, either from application files or data files. For example, a
user may walk into a space, merge their data references into the space, and launch from the
browser a presentation that they have imported from their home file server. Since there is
a choice of displays on which to view applications, simply double-clicking an application or
data file is not sufficient. We allow the user to select from a choice of application config-
uration descriptions (ACD), such as “use all displays”, “use right side of room”, “use left
side of room”, and “use this display” by right-clicking a data source.\(^1\) Since application
configurations depend on what resources are locally available (such as displays, speakers,
\(^1\)ACDs are scripts that allow us to assemble distributed applications by instantiating components on
different machines and linking them together.
etc.), they are only meaningful in certain locations. We tag application configuration files with meta-data thereby allowing the browser to open a directory to find the available configurations for a particular data type. For example, PowerPoint configuration files are tagged with \textit{acd} == \textit{ppt} and location information, such as \textit{location} == \textit{2401}. Thus, when launching a PowerPoint file, the launcher opens the directory /\textit{acd}/\textit{ppt}/\textit{current};, which contains any relevant configurations with which to launch the data file. Users are able to import their own configurations that become visible in the list of options, which may include running a part of an application on a personal device that they have carried into the room. For example, a presentation may be launched so that the slides are shown on the large wall-mounted displays resident in the space and a component to control the presentation is started on the user’s personal handheld device. The launcher takes advantage of the aggregation feature in the data management system to bring relevant configurations - local and remote - together.

Figure 7.2 shows the list of available configurations to launch PowerPoint presentations in our demonstration room. The first six options are system default configurations and the last is user defined, which has been dynamically added. Therefore, by appropriately tagging configuration files, a user is able to program the system such that the file browser sees the new configuration files. The browser does not know what location it is running in - the system is responsible for determining which configurations are relevant for the context, in this case, the current location.

An enhancement that could be made to this application would be to display the configurations only for the current user that is using the browser, rather than have all user configurations displayed. If a fine-grained location system were available, the system could only display the configurations for a particular user by using the \textit{user}: context to prune out the configurations of other users.
Figure 7.2: User data can be added to the files visible to the browser. Custom application configurations may be added to the list of options when launching an application by appropriately tagging data.

7.2 Attendance Recorder

To determine who is present each week during the seminar, we take attendance of attendees and store the information. Once users are authenticated, an event is pushed into the system that contains the user name. When the attendance recorder (see Fig. 7.3) receives the event, it saves the new user name along with a timestamp so that we can determine the number of times that people have missed the seminar at the end of the semester. We also allow users to be manually entered into the system if they choose not to use the biometric authentication system.

When the attendance recorder is automatically launched, it opens the file /current:/attendance.att. The current: directory is used by the system to find the attendance file that pertains to the current seminar, i.e., based on the seminar name. For example, there are several seminars and meetings that are held during the week; the context is used to retrieve the correct data file. Note that the application remains unchanged each time it is executed, but the correct attendance.att file will be opened because the contents of the /current: directory changes
Figure 7.3: The attendance application uses the environmental context to read and write to the correct data source.

based on the current activity; the application is relieved from naming and locating files.

7.3 Paper File List

On commencement of the seminar, a file viewer is programmed to be launched that allows us to select the papers to display on our large wall-mounted displays. We wish to display only the papers for the current week, but would like to store all past discussed files together on disk so that we may access them if necessary. This file viewer is similar to our browser, but only shows the directory contents related to the application data and does not have support for navigation. The file system simplifies the selection of relevant material for the active task by pruning out irrelevant application information. We tag the papers for the week with temporal and situation meta-data; files are given a visible lifetime of one week and made available only to the particular seminar. Therefore, if the same space is used for another seminar (with a different name), these files would not be visible. The PDF file list
opens the directory `/type/pdf/current` when started. The system uses the current time and situation to display the relevant files. During subsequent weeks, these files expire and are no longer visible as new files are added. In this way, we are able to easily prepare the space to automatically show the files we are interested in. Since the application may start when nobody is in the room yet, the applications must be able to find the relevant information, so that human intervention is not required.

7.4 Handheld GIF Viewer

Users are able to enter our active space with handheld devices that can become integrated into running applications. Once the handheld obtains a handle to the local space (via IR beacon), it registers itself and becomes a new execution node that can host application components. For example, a handheld can be used to run components that control a PowerPoint presentation or receive the slides. In the case of viewing slides on the handheld, data transformation is necessary to accommodate the small form factor and lack of native presentation viewer. In Fig. 7.4, a PocketPC has gained access to a PowerPoint presentation as GIF objects. A generic GIF viewer is used to display the images, rotated for better screen layout. The application chooses which format it desires and specifies the dimensions to fit the device characteristics, e.g., graphic context.

Viewing slides on the handheld consists of first attaching to the presentation application running on an execution node in the room, opening the source data file (.ppt) in GIF format, and then receiving notifications by the application when to retrieve the next slide. The system converts the data source (originally in PowerPoint format) on-the-fly for the application. When retrieving data objects, additional properties can be sent with requests, such as specific dimensions of the image objects to accommodate the form factor of the device. While viewing PowerPoint presentations on a handheld is not new, the novelty here is that our file system interface supports the operation of format conversion. The file system API provides
Figure 7.4: The file system dynamic data types allows applications to access data in the format that best suites their needs.

a uniform interface for accessing different data types (bytes, GIF, JPG, etc.) [HBCM01] and content conversion is transparent to the application. By providing content conversion facilities within the file system, the applications remain relatively simple and the application can be developed in less time.

7.5 Music Jukebox

As described in Section 4.4, users may access their personal data from the local namespace by merging data references from a home/domain server or by adding them directly from a handheld computer. We have implemented a third method that allows files residing on the user’s mobile device to become available to the current space. We have added an item to the floating menu when a user “right-clicks” on any file from Microsoft Windows Explorer, as shown in Fig. 7.5.

By selecting the “Add To Current Active Space” menu item, a reference to the file server running on the mobile device (e.g., laptop) is merged into the local namespace and the data is tagged with type meta-data (based on the file extension). We have a music jukebox running in our active space at all times that users can use to listen to songs available on their laptop using the enhanced sound system available in the surrounding space rather than the weak
laptop audio speakers, as shown in Fig. 7.6. Our music jukebox contains a songlist that displays the available songs that may be played. This list of songs is obtained by reading the contents of a directory that represents all music files in the system (i.e., all files tagged as music). The songlist registers for changes in the namespace and receives an update when new data references have been added to the local space. When such an update is received, the songlist refreshes itself by re-reading the directory entries, which now also contains the new music files. Songs are retrieved from the laptop on demand and used by the jukebox application running in the space. In this way, a laptop and its file store can be integrated into the local environment and the application is allowed to remain simple; the system deals with adding the user data, directing the data the the application, removing any data not meant for the current context, and notifying it when there are changes. We can perform similar operations for any data type; we do this routinely for PowerPoint and PDF files. When a user leaves the space, the files are automatically removed from the space and songlist. Note that data references are added to the space, i.e., the data is not copied to the space. Copying
the data to the space would be prohibitive for large amounts of data (e.g., a large number of music files), all of which may not be used.

![Diagram of file system and namespace change channel]

Figure 7.6: Files are retrieved from a laptop after a temporary mount pointing back to the laptop file server has been added to the mount server.

## 7.6 Greeting Engine

*Gaia* includes a voice engine that accepts text strings, translates the text to speech, and sends it to the room speakers. This service is used for audio feedback, to notify users of events, and to greet people when they enter a space. The greetings that are spoken can be customized based on the current context, such as part of day or the current activity. For example, the space may detect that Bob has entered the local space and says “Good morning Bob”. Depending on what part of the day it is, different greetings can be issued, such as “good afternoon” or “good evening”. The greeting engine simply reads the file `/current/greeting.txt`. Each greeting is stored in a different file, but the links to each are all named `greeting.txt` and the correct one is activated based on the context. For example, to
customize the greeting engine to use part of day (e.g., morning, afternoon, or evening), the
greeting files (e.g., `greet-morning.txt`, `greet-afternoon.txt`, and `greet-evening.txt`) are tagged
with the context `partofday == <part of day>`, where `<part of day>` is `morning`, `afternoon`, or
`evening`, respectively.

Note that the greeting engine could be customized for different types of context, such as
activity. For example, if a meeting is active, the greeting may say “Welcome to the meeting,
Bob”. The key point is that by removing the context knowledge from the application, it
becomes much more flexible to customize applications because we can now change their
behavior by simply changing tags in the file system rather than making changes to the
application.

### 7.7 Administrative Examples

In this section, we will present several examples of how the system may be used by an
administrator to configure applications. We will take three of the above examples and
explain the steps required by the administrator to setup the system.

#### 7.7.1 Gaia Data Browser

When launching an application from the Gaia Data Browser, the browser opens the context-
mode directory `/acd/:<file extension>/current` to find available configurations. Suppose that
we have a configuration file called `main.lua` that starts a PowerPoint presentation that we
wish to make available to a space called 2401. We must tag the configuration with metadata
of `acd == ppt` and `location == 2401`. We can use the Gaia Data Browser itself to attach
the metadata by using the following steps:

1. Place the `main.lua` file into a file-mode directory accessible from CFS.

2. Switch to context mode by selecting the top left button with the globe icon.
3. Go to the root directory by repeatedly pushing the “up directory” button until the current directory is “/”.

4. Right click anywhere in the explorer to bring up the floating menu.

5. Select “New Folder...” and enter acid: in the box if the directory does not already exist.
   The new directory should appear.

6. Change the current directory by double clicking on the newly created directory.

7. Repeat the previous three steps to create the remaining directories, e.g., ppt, location:, and 2401.

8. Switch to file mode by selecting the button second from the left on the top row (i.e., with the file icon).


10. Copy the file by selecting it and either pushing the copy icon or hitting ctrl-c.

11. Switch back to context mode.

12. Paste the file by either pushing the paste icon or hitting ctrl-v.

### 7.7.2 Attendance Recorder

The attendance recorder opens a single /current:/attendance.att file that contains the information of participants. There may be several files with the same name; however, the file system will activate the correct file depending on the current activity. We create separate files for each activity with different names, such as attendance-seminar-fall03.att, and then create a link to it called attendance.att. Since only one file is opened at a time, it is not necessary to attach any properties to the file; context information is sufficient. The recorder runs during different activities, whose context is set by the calendar application. The calendar
sets the *situation* context for a duration of time, so that *situation* metadata must be attached to each file to activate them. The following steps can be used to add a new attendance file when a new type of activity is scheduled:

1. Create a new attendance file with any name in a directory accessible from CFS.

2. Switch to context mode and go to the root directory.

3. Create the directory `/situation:/<situation>`. `<situation>` should match what the calendar sets the situation as.

4. Switch to file mode.

5. Navigate to the attendance file and select it.

6. Copy the file.

7. Switch back to context mode.

8. Paste the file.

9. Select the file and right click.

10. Select “Rename...” and enter *attendance.att*.

This same sequence of steps is used to configure the greeting engine. The only changes are that the *situation:* directory is changed to *partofday:* and the links to the greeting files are renamed to *greeting.txt*.

### 7.7.3 Paper File List

The paper file list looks for PDF files relevant to the current context and displays them in a list. We can bind these papers to different activities (e.g., a seminar) for a certain duration of time (e.g., a week) by tagging the files with the *time:* built-in context and the *situation:* context together as follows:
1. Place the PDF file in any directory accessible from CFS.

2. Create the /type:/pdf/situation:/<situation>/time:/<start>:<end> directory, where <start> and <end> are in the form yyyy-mm-dd, e.g., 2003-3-7:2003-3-14. Parts of this directory may already be created. Adding the directory under time: may only be required.

3. Switch to file mode.

4. Navigate to the PDF file and select it.

5. Copy the file.

6. Switch back to context mode.

7. Paste the file.
Chapter 8

Evaluation

Ubiquitous computing systems are difficult to evaluate since they often stress new functionality and ease of use over pure performance. In order to evaluate the file system, we use three methods. Our goal is to show that 1) programming complex tasks is simplified, 2) new operations are enabled, and 3) response times of the system match user expectations. In this evaluation, we describe how the system is able to make applications easier to construct and show the performance and usability of our system for both people and applications during everyday normal use.

8.1 Application Development

The programming wrappers for the file system allow developers to take advantage of the low-level functionality with relatively few lines of computer code. Here, we give some concrete numbers in terms of amount of effort required to use the system compared to writing the functionality into each application. We take as examples 1) the GIFContainer with data format transformations of ppt → gif and 2) the BitmapContainer with transformations of ppt → gif → pixmap → bitmap. Later in this chapter, we evaluate the speed of these transformations.

The first transformation requires the system to load one module that implements the conversion from PowerPoint objects to GIF objects. The interface is implemented with
the *PowerPointContainer*, which consists of 270 lines of developer code (not including all interface code, name resolution, header files, and generated code from the IDL compiler). As shown in Section 6.3.4, the same operation can be achieved in 10 lines of code by using the container wrappers. In addition, use of the container wrappers hides the complexity of the C++ interface to the COM object that performs the conversion process.

For the conversion chain described in point 2 above, the system loads three containers, including a *PowerPointContainer*, *GIF2PixmapConverter*, and *Pixmap2BitmapConverter*. The combination of these three containers amounts to approximately 460 (270+90+100) lines of code, but still allows the developer to access the functionality in 10 lines of code.

One of the goals of *Gaia* was to provide a simple abstraction so that a room and its devices may be easily programmed. The development of the *Gaia* wrappers has made this possible, greatly simplifying program development. The file system wrappers contribute in this respect as well. The wrappers for the file system are arguably the most complex of the *Gaia* wrappers, because they provide much more than simply hiding the details of the CORBA communication infrastructure; they provide the mechanisms to resolve paths, discover and cache available mount points, and contact remote file servers.

### 8.2 Enabling Technology

Probably the most important aspect of the file system is what it enables. Its functionality allows users to have data move with them, applications to be automatically launched, and data format conversion, which has allowed us to build applications that would otherwise have been difficult or not be possible to build. Some of these applications are described in Chapter 7.

Without the format conversion capabilities of the system, it would have been much more difficult to construct an application to view PowerPoint presentation on a handheld. In addition, the container framework makes it fairly simple to add new converters to the
system. The ability to inject user data references into the namespace of a room has allowed us to easily transport our data around with us. Use of the authentication service makes this process invisible to the user. Finally, the system allows users and applications to associate meta-data with files, which in conjunction with the current context active in a space, has enabled us to write applications that may be automatically launched and are able to easily find the data they require. For example, the attendance application can access its data in a couple lines of code, where the system provides the complex task of discovering where the data is located and presenting it to the application based on the current context.

8.3 Performance

In this section, we evaluate the response time of performing mount server queries, creating and destroying context directories, and copying and removing files to/from context directories. We show that the performance is able to meet the requirements of our applications.

8.3.1 Queries

We have implemented a shell program and graphical browser (described in Section 7.1), that allows us to manipulate the file system hierarchy. While the graphical browser is used on a daily basis to attach context and launch applications since it only requires us to push buttons on our large wall-mounted touch panels, the command line shell more accurately measures the performance of the file system components since the graphical overhead is eliminated. We have not optimized the code to a great extent and our current implementation runs in debug mode. When using the graphical browser, response times are entirely dominated by the Windows graphic libraries, resulting in times orders of magnitude greater (~40 msec/operation). Therefore, response times for operations initiated by human users is comparable to those experienced with typical desktop graphical directory navigation tools.

Figure 8.1 shows our experimental setup. All machines in the experiment are 1.5 GHz
Figure 8.1: The performance numbers were obtained from data from our active space that we use on a regular basis.

Pentium-4 PCs with 256 MB RAM running Windows 2000, service pack 2. All nodes are connected through a 1 Gbps Ethernet switch. This setup is indicative of how our active space is configured for everyday use and therefore reflects actual response times experienced in normal operation. The shell communicates with a distributed file server, which loads a module that represents our context directories and is responsible for querying the mount server, retrieving mounts, fetching directory entry objects from the remote file servers, and returning them to the shell. All experiments were run 1000 times and results were averaged to obtain the time for a single operation.

The first experiment involved simply measuring the time to perform queries at the mount server in order to investigate scaling issues. We increased the number of mount points that were available in the mount server from 10 to 50. We performed queries with a single tuple (i.e., /location:/2401), with all mount points containing a single meta-data tag. We varied the number of mount points that matched each query from 1 to 4. The results are shown in Figure 8.2(a). We expected times to increase linearly with the number of mount points and matches, which the experiments corroborated. Next we issued queries with increasing
number of tuples. When a query of \( n \) tuples was sent, we fixed each mount point to contain \( n \) meta-data tags, so that matches could be obtained. \( n \) was varied from 1 to 4. The results are shown in Figure 8.2(b). Again, results vary linearly with number of mount points and number of query tuples. As expected, scalability is related to the number of mount points stored in the mount server. Assuming the number of mount points that each person adds to the system is limited, the scale of the system is loosely proportional to the number of people that fit in a room.

![Diagram](image)

(a) Results for 1-tuple queries. (b) Results of multi-tuple queries.

Figure 8.2: Response times increase linearly with the number of mounts.

In the next experiment, we determined the time for various context mode operations in order to measure the end-to-end response time of the overall system, as shown in Table 8.1. We used the mount points available in our active space that have accumulated over time, which number approximately 50. The first of these experiments listed the contents of the root directory, which contained 8 entries. In this case, no data file servers are contacted other than the initial module (the center file server in Fig. 8.1), since the virtual directory is constructed entirely from the contents of meta-data tags in the mount server. Each listing took 0.711 msec. Next, we listed two directories that contained files, the first which required that a single file server be contacted to retrieve 4 items and the second which required contact
with two file servers to retrieve 4 items from each. Results for each listing was 1.720 msec and 2.825 msec, respectively. Finally, we tested the time it takes to create/remove context directories and copy/remove context files. Creation and destruction took 8.948 msec and 36.669 msec, respectively. The main reason directory destruction is more time consuming is that the system makes sure that the directory is empty before it removes it, therefore, adding an extra directory access. The time for copying and removing a file to a context directory was 11.303 msec and 9.021 msec, respectively. Overall, the system is responsive for human users and applications are able to locate and transfer data at an acceptable rate for those that we have developed.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>List Root</td>
<td>0.711</td>
</tr>
<tr>
<td>List Directory (1 fs)</td>
<td>1.720</td>
</tr>
<tr>
<td>List Directory (2 fs)</td>
<td>2.825</td>
</tr>
<tr>
<td>Create Directory</td>
<td>8.948</td>
</tr>
<tr>
<td>Remove Directory</td>
<td>36.669</td>
</tr>
<tr>
<td>Copy File</td>
<td>11.303</td>
</tr>
<tr>
<td>Remove File</td>
<td>9.021</td>
</tr>
</tbody>
</table>

Table 8.1: Measured times for context mode operations.

In using our system, we experienced a situation where performance can be significantly degraded. If a user enters a space and a file server hosting their personal data cannot be reached, the system hung until the server could be reached. However, sometimes we had outdated mount points or a particular laptop was not attached to the network. In order to alleviate this problem, we decreased the timeout when a file server could not be reached down to 2 seconds. However, one can imagine that different types of applications will require a different response to inaccessible servers depending on how important the information is to the application. For example, an MP3 player may skip any music files that it cannot retrieve and continue. However, other applications, such as the attendance application, may want to wait until the file server is reachable again. These observations dictate required design
changes to the system and are areas of future research.

8.3.2 Format Conversion

Our system performs on-the-fly data conversions for different data types. This section evaluates the speed at which such conversions can be performed for several transformations. The first transformation is from PowerPoint format to GIF format. This conversion is used in our PDA presentation viewer application described in Section 7.4.

To provide this conversion, one module is used, which wraps the Microsoft PowerPoint COM object that contains methods to export slides to file in a certain format. Therefore, when a client opens a PowerPoint data source in GIF format, the module is loaded, and requests for objects result in the COM object writing out a slide in GIF image to disk with specified dimensions, reading the image data back into memory, and sending the image to the client. In determining the speed at which this operation can be performed, we used slides with three different characteristics: text only, graphics and text, and images and text, labeled as Text, Graphic, and Picture, respectively, in the tables below. These different types produced GIF images of increasing size. We measured the time it took to retrieve objects of each type with different dimensions over the wired Gigabit Ethernet. Times were taken as an average over 10 identical objects. The results are shown in Table 8.2 and do not include the time to uncompress the image at the client side.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Text</th>
<th>Graphic</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>80x80</td>
<td>0.0200</td>
<td>0.0450</td>
<td>0.0630</td>
</tr>
<tr>
<td>160x160</td>
<td>0.0251</td>
<td>0.0541</td>
<td>0.1041</td>
</tr>
<tr>
<td>240x160</td>
<td>0.0281</td>
<td>0.0631</td>
<td>0.1181</td>
</tr>
<tr>
<td>352x240</td>
<td>0.0360</td>
<td>0.0831</td>
<td>0.1943</td>
</tr>
<tr>
<td>640x480</td>
<td>0.0771</td>
<td>0.1812</td>
<td>0.4797</td>
</tr>
<tr>
<td>800x600</td>
<td>0.1101</td>
<td>0.2434</td>
<td>0.6099</td>
</tr>
</tbody>
</table>

Table 8.2: Time in seconds to retrieve a GIF image object from a PowerPoint source.
For PocketPC handhelds, the typical screen size is 240x160 pixels. For small image sizes, which are typical, slides containing graphics and text took slightly more than .05 seconds, which we found to be acceptable. Based on our experiences with the system, the wireless network dominated the time to display images on the handheld. However, even with wireless networking time considered, images were able to be displayed with minimal delay.

In the next evaluation, a chain of conversion modules were connected together to convert PowerPoint slides to bitmap objects (i.e., ppt → gif → pixmap → bitmap). Such a conversion path could be used by a device that only supports a limited number of colors, such as phone or PalmPilot, or whose processing power is too small to uncompress GIF formatted objects.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Text</th>
<th>Graphic</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>80x80</td>
<td>0.0210</td>
<td>0.0491</td>
<td>0.0641</td>
</tr>
<tr>
<td>160x160</td>
<td>0.0311</td>
<td>0.0471</td>
<td>0.1162</td>
</tr>
<tr>
<td>240x160</td>
<td>0.0400</td>
<td>0.0641</td>
<td>0.1342</td>
</tr>
<tr>
<td>352x240</td>
<td>0.0681</td>
<td>0.0771</td>
<td>0.2303</td>
</tr>
<tr>
<td>640x480</td>
<td>0.2043</td>
<td>0.3025</td>
<td>0.6069</td>
</tr>
<tr>
<td>800x600</td>
<td>0.3054</td>
<td>0.4396</td>
<td>0.8132</td>
</tr>
</tbody>
</table>

Table 8.3: Time in seconds to retrieve a Bitmap image object from a PowerPoint source.

The results show that the inclusion of additional modules in the conversion chain only slightly increased the amount of time to fetch objects. These results indicate the a majority of the time is used in the PowerPoint COM object to generate the GIF image from the original PowerPoint slide.
Chapter 9

Lessons Learned

In the following sections, we discuss some of the experiences with the system and what we have learned from using it. These observations have resulted in design and configuration changes and have exposed where possible future improvements could be made. We describe our experience with the overall system and later discuss some file system specific issues.

9.1 Experiences with a Seminar Room

One goal in our active spaces project was to build a real system that we could deploy, use, and experiment with. Once enough of the system was constructed, we started using Gaia during our weekly seminars in room 2401 in the Digital Computer Lab building of the Computer Science Department. A seminar typically consisted of discussing two papers that had been read prior to the group meeting. A schematic of the room is shown in Fig. 9.1. The wall mounted displays and table-top touch screens were used to display our papers and people sat around the conference table. There was generally little movement (i.e., walking around or interaction with the large displays) during the seminar. During regular use of the system for the past semester, we have discovered what people have found to be useful and not useful. The Gaia infrastructure allowed several applications to be automatically started when the seminar began, allowed people to import their own data into the space, and to control applications from any machine in the room. A typical seminar proceeded along the
Figure 9.1: The seminar room contains a central conference table with wall mounted and table displays.

The following lines:

- **Scheduling.** The seminar was scheduled at the beginning of the semester. Scheduling the seminar involved setting the context that will be active, the duration of time that the seminar is to take place (1 hour in our case), and what script to launch on commencement.

- **Setup.** A small amount of setup was required each week. This involved downloading the files into the file system and tagging them with type, situation, and time meta-data.

- **Launch.** When the seminar time arrived each week, the scheduled script was launched that started our applications, including the attendance recorder and a PDF viewer. The PDF viewer started a file list that contained the (2) files for the week, setup in the previous step. From this list, we could place electronic versions of the papers on various displays.

- **Authentication.** Students authenticated with the system via fingerprint detector. From
this information, the attendance recorder stored who was present during the seminar.

- *Control.* When the PDF application was launched, the controller component was instantiated on the machine from which it was started, which was usually a wall mounted display. We usually moved the controller to a display on the table or to a handheld so that it would be more convenient to use.

After using the system for a semester, we noticed where improvements could be made to make the system easier to use. For example, we added audio feedback during the authentication process to allow users to know when they had been successfully logged in or when authentication failed. This helped considerably, since when using the finger print detector, it was unclear how long one had to leave the finger on the device or when a retry was necessary. Aural feedback was also used when new devices were added to the space. Before this was included, we had to bring up another tool to see if the device had successfully registered in the space. Problems with the wireless networking due to weak signal strength and electromagnetic interference in some parts of the room made this extremely useful, as devices often could not register with the space. We also noticed that the wall mounted displays were at a distance from the central conference table around which people sat, making the displayed papers difficult to read. We then decided to move the papers to the two displays located on the table. While this helped to make the papers easier to read, we found that most people did not read the papers from the displays. When pointing out a passage in a paper during discussion, we would scroll the displays to the correct page. However, most people would look at the hard-copy that they brought with them, since they also had marked them up with notes. Part of the problem lay in the fact that we had no easy way to annotate the electronic documents with our software.

A system such as *Gaia* is probably better suited as a presentation or collaborative design room. A seminar is generally a “read-only” scenario, where users simple discuss a papers that has already been read off-line. The automatic logging of user attendance and the ability
to bring laptops into the space did prove beneficial. However, the integration of devices and data are probably better suited for collaborative sessions, where the face-to-face physical collaboration can be enhanced by the presence of digital devices.

Several aspects made the room less useful than we had anticipated. However, these deficiencies could be fixed with added software support. The first impediment was the lack of simple user interface. In terms of usability, an intuitive graphical interface (i.e., space desktop) that could be used to easily redirect resources and move application components around the space would be greatly beneficial. The second aspect was the lack of support to synchronize displays of all file formats. For example, we currently do not have support to view Microsoft Word documents in a synchronized manner. While the basic building blocks are in place, additional applications built upon this foundation would make the system much more useful.

9.2 Drag-and-Drop Support

A common function that we typically performed was attaching attributes to files in preparation of the seminar. For our PDF file viewer, we attached attributes with time, situation, and type information, which were represented as different directories in the virtual hierarchy. This allowed the PDF file viewer to automatically show the files for the given week. While our graphical browser made the operation easier, we could exploit redundant information already available in the room to perform the same operation in fewer button clicks. The situation and type attributes are used every week and are therefore only created once at the beginning of the semester. However, when we added new files each week, we had to create a time context directory, which required us to type in the valid time range. We typically performed this operation at the large touch panels, where typing was tedious and should be reduced to a minimum or eliminated. However, all the information that was required already existed in the calendar application. A method to simplify the task of adding context to the
seminar files would be to drag-and-drop the the files onto the calendar application. Since the situation and time information is available in the calendar and the type information can be inferred from the extension of the PDF file, a mount point could be automatically generated through the calendar. This would greatly simplify the operation of associating our PDF files to our weekly seminar.

9.3 Setting Context

When the attendance recorder application started, it opened a file which contains the historical attendance information. The context defined by the space dictates which file is available for the application. As described above, different context will allow the application to run unchanged for different tasks. However, a problem can arise when no context is defined in the space. Since we manually set the context via the calendar application, no context is set when nothing is scheduled. When no context is defined, all available attendance storage files are available under the /current: as a result of the file system matching algorithm. There are two solutions to the problem. The first is to always set a context, even if no activity is taking place. For example, the context of situation == none could be set for empty tasks. This could be the default context or could be set by the calendar application when nothing has been scheduled. A second solution would be to allow the application to specify that it is interested in files only if they have a particular context type attached to them. For example, the attendance application may request attendance files only if they have any context of situation attached to them. Therefore, the system would not return any files if the situation context was not active. This may require small changes in the directory syntax, such as allowing wildcards, i.e., /situation:/*/current:.
9.4 Parallel Aggregation

In using our system, performance could be degraded when mount points referenced non-existent machines, i.e., they are not connected to the network. In such cases, the system has to timeout when opening directories that reside on these machines. This problem can be particularly egregious for context directories because they often aggregate the contents of multiple file servers to appear as one source. Therefore, multiple inaccessible file servers can result in large delays. However, a change in the implementation could reduce this delay to that of a single server. By contacting file servers in parallel rather than sequentially, the directory entries could be retrieved as they become available, greatly reducing wait time and increasing the responsiveness of the system.
Chapter 10

Related Work

The concept of ubiquitous computing was born at Xerox PARC in the late 1980’s. The original work laid out the vision for a new type of computing, one in which computing becomes “invisible” and fades into the background of consciousness. The concept of making applications context-aware was another novelty of the research at PARC. Today, ubiquitous computing is a hot topic in research circles and there is a large body of literature that relates to our work. In this chapter, we describe the research most relevant to ours and how our work differs from others. We begin with a survey of related work in ubiquitous computing spaces and later discuss work directly related to data access.

10.1 Ubiquitous Computing Spaces

Early work in ubiquitous computing systems developed custom software for specific environments. Recently, more attention has been given to developing more flexible systems software for ubiquitous computing spaces. Our approach has been to provide a complete system implemented via an abstraction layer so that a space and its resources can be programmed as a single entity. By allowing developers to program to the abstraction layer, applications become portable across different spaces and are not tied to a specific environment. While the approach taken in Gaia is different from much of the related work, we discuss several of the important projects since they have been very influential to our research.
One of the first implemented systems introduced the Active Badge [WHFG92, ST93], which comprised of a small device that periodically sent out IR signals that were detected by a sensor network attached to a personal computer. The badge was worn by a group of researchers at the laboratory to track their location and was used to route telephone calls to them based on where they were located. Another of the early seminal works was the ParcTab experiment [WSA+95]. The project consisted of the development of several devices, the ParcTab, ParcPad, and Liveboard, each device increasing in size. The goal of the project was to investigate mobile computing, context-sensitive applications, and the design of hardware devices to support mobility and context. The Tab was a small wireless handheld device that could interact with standard Unix systems that contained three control buttons and touch screen. A graffiti writing interface was also available to enter text. The Tab was used to access location-based information, interact with other devices (such as lights and electronic whiteboards), along with a host of other mobile applications. The Pad was similar to today’s TabletPC and Liveboard similar to an electronic whiteboard.

Georgia Tech has produced some important work in context-aware environments and applications. Much of the work involved building systems for real environments to study the human response to such systems. Classroom 2000 [AAF+96] had the goal of deploying a smart classroom in a teaching environment. The project included three phases that included pre-production, in-class, and post-production to help in note-taking, indexing, and search of class material. Extensive user studies showed the benefits such a system brought to a learning environment. Ongoing research is investigating ubiquitous computing in the home environment as part of the Aware Home project [KOA+99]. This project built a home containing two living spaces (one for demonstrations) and is interested in what applications could best assist home residents and what technologies are required to build these applications.

The Cooltown project [Hew, KB01] from HP Labs uses location to scope the information on the WWW to what is important for the current location of a person. Cooltown strongly advocates the used of web infrastructure as the middleware to access content. They use the
concept of “web presence” to limit what information is presented based on user locality. Small
devices situated in physical environments send local URLs that point to locally interesting
material via beacons. Beacons are picked up by mobile devices that are carried by users and
are used to display information for the current surroundings. While web protocols are useful
in that they are the defacto standard and are ubiquitous, they are limited in the amount of
interaction that is possible. Highly interactive applications are difficult to map onto the web
model.

Easyliving [Mica, BMK+00] from Microsoft Research is developing smart environments
in which the diverse set of I/O devices can be interconnected in the ways that best suite
present users to provide a single coherent system. Using the preferences and location of a
user, applications can move around a space using various devices. Their work is involves the
middleware to connect the devices, a geometric model to locate entities within a space, and
tracking of entities via stereo-scopic cameras. They developed special middleware, called
InConcert, that provides asynchronous remote procedure calls to accommodate the dynamic
environment. In addition, their remote invocations use a dynamic binding mechanism in
which objects are located on each invocation to support the mobility of system components.
A central concept in Easyliving is the geometric model that is used to determine the plac-
ement of objects in a space. This model allows for a more natural association between the
digital and physical identity of an object so that humans can more easily manipulate them.
The geometric model can be updated with information from the cameras that are able to
track the location of moving objects. Some of the goals of Easyliving are similar to those
of Gaia. However, Gaia has additional aims, including the partitioning of applications and
security services.

The i-Land [SGH+99] project is developing infrastructure and hardware devices for co-
located collaboration. The i-Land infrastructure is constructed of a layered architecture
called Beach [Tan00]. Beach supports multi-device interaction, composite components, dy-
namic configuration, and adapted visualization. The core of the system supports service
management (transactions, replication, and persistence) and a set of base services, such as events, sensor management, and display context. Higher layers consist of application building blocks, application composition, and task modules consisting of several applications. This software has been used to integrate several “roomware” [SGH98] components, which included 1) DynaWall, a large collaborative touch-sensitive display, 2) CommChair, a chair for mobile users that is able to dock mobile devices that can interact with room elements, such as the DynaWall, and 3) InteracTable, a digital display table with arbitrary orientation. The goal of the project is to provide support for group activity and provide mechanisms to make annotations that can be associated to activities.

The Interactive Workspaces project [FJHW00, JFW02] at Stanford is building infrastructure and applications for collaborative meeting rooms and is influenced by the i-Land project. These workspaces, called iRooms, are populated by a variety of large wall-mounted displays and input devices. Their focus is on co-located collaboration in contrast to distributed collaboration. Areas of focus are the use of several devices simultaneously, the movement of applications, and the connectivity of input and output devices. The iRoom consists of the Interactive Room Operating System (iROS), a meta-level operating system that coordinates local devices. iROS is composed of three subsystems, mainly the Event Heap [JF02], Data Heap, and iCrafter [PLF+01]. The Event Heap is a tuple space used to distribute events and decouples components in the system. They have added expiration times to events (which are stored in the Event Heap for a short duration), so that obsolete events eventually are removed from the system. The Data Heap is a service that allows applications to store data that can be used in the local environment. Data items can be tagged with properties, similar to our file system. The Data Heap uses a WebDAV [Web] server to store and share information, which can later be retrieved and used by applications. The structure of the Data Heap makes some operations difficult, such as importing a large number of music files to a space, since the data must be copied (rather than referenced) to the data server. The final component, iCrafter, provides support for interface generation, and service advertisement
and execution. Much attention has been paid to fault tolerance and robustness, striving for a self-healing system in the face of component dynamics and failure. Similar to Gaia, they identify the need for an “operating system” for physical spaces to support applications. The services provided by the iROS infrastructure are a subset of those provided by Gaia. For example, Gaia provides a more comprehensive data access service, security, entity discovery, and an application framework.

The Virtual Home Environment (VHE) model [EUR] proposes an architecture where mobile users may access their environment (e.g., services) from different locations and devices. The model considers device and network heterogeneity with the goal of presenting a consistent look and feel to services. Our model is more related to an operating system, by treating a space as a programmable entity to assist in the development of interactive ubiquitous applications.

The University of Washington has developed an infrastructure called one.world [GDH+01]. The design of one.world is motivated by the new constraints imposed by pervasive computing on distributed systems, such as dynamism and frequent unavailability of devices. These factors drove the decision to create a development platform based on asynchronous method invocations. However, this architecture imposes restrictions on how applications are constructed. While the asynchronous architecture is useful for handling unavailable components, the model complicates the programming of applications. Labscape [AGH+02] was built on the one.world platform, but did not strictly adhere to the programming model.

MIT’s Intelligent Room [Bro97] project is constructing a smart room that makes heavy use of vision technology to self-calibrate the system. Recognizing that futuristic rooms populated by many sensors will be difficult to configure (i.e., the location of devices), they incorporate techniques of image recognition to determine the layout of devices within a space. The room architecture is composed of three layers, the bottom of which monitors the interactions of entities within the space, the middle layer which consists of agents that provide an interface for interacting components, and the top layer that consists of application agents.
From the Intelligent Room came the Metaglue [CPW+99] system, an agent-based software infrastructure for handling resource management within the system. Metaglue consists of extensions to the Java language to support agent configurations and state management. The system allows software components to ask for resources based on a description, from which the system can find an appropriate resource (if no conflicts result), start the resource, and allow the component to interact with it. This functionality allows the system to operate in rooms with different resources, requiring no changes to the applications. This is similar to our idea of mapping applications to different spaces based on available resources.

The Portolano Project [EHAB99] at the University of Washington is constructing a data-centric platform for mobile and ubiquitous computing. The focus of the project is on dynamic user interfaces, agent technology, resource discovery, data-centric networking, and distributed computing. The ideas build on existing technology, but incorporate many of the issues that arise in mobile ubiquitous environments. Such issues include intermittent connectivity, format conversion, quality of service, context to anticipate user actions, and device heterogeneity. The problems that are explicitly tackled by Portolano are very similar to those addressed by Gaia.

Jini technology enables heterogeneous devices equipped with a Java virtual machine to discover services in physical spaces [Wal]. Devices may register themselves with the Jini lookup service and once registered, other devices may discover them and immediately use their services. Using the code mobility of Java, custom user interfaces or application may be sent to client devices to allow interaction with services or resources. The main focus of Jini technology is on the discovery and use of services. Our infrastructure provides these capabilities, but included additional services to support ubiquitous applications in an active space. However, we could have implemented Gaia using a technology like Jini.
10.2 Context-Aware Data Access

Early work in integrating context with file access was investigated as part of the ParcTab project. The Tab allowed access to files that were meaningful to a particular location [SAW94]. As users moved between office spaces, the file browser would change to display relevant data. While they only considered location in their file system, this seminal work was important in establishing the relevance of context in data access and application adaptation. Development of location-based tour guides and remembrance agents incorporate the location of the user to present relevant information. MemoClip [Bei00] triggers notifications based on where a person is located. MemoClip consists of a small hardware device which contains a small database that holds event-location pairs. The device polls surrounding beacons for location information and notifies the user when the location is associated to some event. CybreMinder [DA00] provides similar capabilities, but a richer set of context can be associated to the reminder information, such as time, location, weather, stock price, co-location, etc. CybreMinder has several delivery mechanisms, such as e-mail, local display, SMS, or printer. These projects have investigated making certain data available in a certain context, but were specific to particular applications. Our goal was to look at building a generic infrastructure.

The Stick-e document framework [Bro96] describes information in SGML format that includes data and context information. The framework includes several modules that allow Stick-e documents to be created, managed, triggered, and shown. When a specified context matches an available document, a trigger makes the data available. This framework has some similarities to our system. However, our approach is different in that we do not generally trigger actions based on contextual changes, but rather keep users “in the loop” and allow them to choose what actions to take. For example, we allow users to inject personal application configurations into a space, but the user is responsible for picking a desired configuration, resulting in a pull-based system. Context-aware retrieval (CAR) [JB02] has been studied with regards to presenting users with important information based on their current
context. A proactive context-aware cache was developed in which the user’s (future) activity is used to help warm a store of anticipated useful documents. The context information is gained from the users location and history. Our system has not considered caching of data; rather we are interested in allowing a space to be programmed so that data can be directed at certain applications and we use context to determine when that information is made available. The Personal Server [WPD+02] is a small data server that can be carried around with a user. The device offers no graphical display, but intends to use larger displays that are available within a local space. The system is housed in a small case that can be worn by the user and uses a WebDAV server to access personal data. Our mobility support allows users to carrying data references on handheld devices that are merged with the infrastructure. However, our design also allows mobile devices to run a file server so that local data can be integrated into the local environment, as shown with our Windows Explorer extension.

Research in tangible interfaces has proposed tying digital information to physical objects, which can trigger some action (e.g., file transfer) when they are discovered by a new environment [FIB95, IU97]. Attaching digital information to tangible objects can ease the way in which information is made available to a new environment. We expand on this idea by treating the user as the physical object that triggers the addition of information into a space, implicitly linking storage to a user. In addition, physical objects, such as handheld devices, may actually contain the information, rather that just being linked to it.

### 10.3 File Systems

There is a vast body of work in file systems research and much of the distributed file systems research has focused on issues orthogonal to the ones we are addressing in this work, such as performance and consistency. In this section, we describe several file systems that are most related to our work, which have involved the study of alternate structures and naming schemes.
The syntax for our virtual file hierarchy is similar to the Semantic File System [GJSJ91]. The system indexes data sources when files and directories are created and updated and the path components are used to provide associative data access. Data extraction is performed using transducers, where different transducers can be applied to provide the user with alternate views of data. The system also employs a query mechanism for finding information using attributes. The system can be extended through the use of user-defined transducers to customize data retrieval. Information is organized based on some characteristic contained within the data, for example email messages can be organized by sender, where a directory is created for each sender and a link to the mail messages corresponding to each are placed in the appropriate directory. Our system uses meta-data to organize the data and uses contextual information to drive the applicability of information. While our work is basically different from the Semantic File System, their research developed fundamental concepts of virtual file systems. Gopal et al. have extended the ideas of the Semantic File System to include semantic directories to group related material [GM99]. Their system populates these directories with symbolic links to information gathered from queries to the system that match data which is of similar interest. The results of queries can be altered by adding or removing links to semantic directories, thereby allowing semantic directories to act like regular directories that can contain files. Our system differs in that we use meta-data to define how information is grouped, rather than allowing a system search to determine what information is aggregated into a directory. The Prospero File System [Neu92] constructs a virtual namespace based on views that are generated by different filters. Filters are attached to directory links and are applied to the link to allow the view of data to be altered. The system is able to group logically related material together in different ways to facilitate data organization. Data can reside on different servers and has been used to organize information on Internet sites, based on search criteria. We have approached the problem from the perspective of applications running in a space. Rather than simply define alternate views of data for users, our organization is targeted at running and automatically launched applications
by tagging information with meta-data. A main difference between our system and the above described file systems is that our system explicitly integrates information from the environment to affect how and when data is made available. A further difference is that our virtual mode directories are not hierarchical, i.e., they may be traversed in any order, whereas the others organize data in an ordered fashion. Since contextual information is typically not ordered, such systems would not suite the type of environment we are targeting. In addition, we have introduced mobile data references and dynamic data types, both considerations that we believe must be addressed in a file system for ubiquitous computing environments.

Presto [DELS99] is a document management system that uses property tags to organize data so that different users may have personalized views of the data hierarchy. They developed a graphical desktop where files could be grouped together and properties could be dropped on the desktop to display the items that exactly matched the active properties. Our system has some similarities to this system in that our attributes act like their properties. However, their properties act as filters, where each added property narrows the list of data matching the property list. We differ from this work in several respects. First, we use implicit environmental context to display relevant material. Second, our queries are different from filtering. In our system, some environmental context may not be relevant to a given application, and we therefore ignore such context; some of our queries would fail to match items in the Presto system. Third, our system is targeted at organizing data for automatically launched applications in addition to users. Finally, we incorporate the mobility of users, allowing them to merge their data into a new space.

10.4 Content Conversion

Work in accommodating heterogeneous devices or networks through content conversion has been studied by several groups. Relevant issues are monitoring of network conditions, proper
developer interfaces, and frameworks that support different types of conversions. We have focused mainly on the developer interface by incorporating the conversion mechanisms into the file system design.

Odyssey is a file system that has been developed to support mobile applications [NFS00, Nob00]. Odyssey defines a metric called fidelity that measures how closely the data a client application receives matches the reference data. Data may have varying degrees of fidelity, which corresponds to different levels of data degradation. The system is composed of war-dens that encapsulate type-specific data knowledge and can also be queried to retrieve data attributes. This concept is similar to our dynamic data type attributes, in which a data source maintains a canonical format, but may be adapted to the needs of an application. Their system focuses on the varying degrees of network connectivity for mobile users and the speed at which applications can adapt to changing conditions. Our system assumes connected service and we focused on the developer interface to access the conversion mechanisms and the framework to add new conversion modules.

The Ninja architecture [GWvB+01] defines four main components: bases, units, active proxies, and paths. Bases are manifested as a cluster of workstations that provide scalability, fault tolerance, and concurrency. Units comprise the myriad of devices that may be connected to the infrastructure. The active proxies provide adaptation of content (similar to our containers), and are the result of previous research in data distillation using the TACC [Fox] model to perform on-the-fly data transformations [FGBA96]. Transcoding data formats was found to greatly increase the performance of certain applications [FGG+98].

The last component, paths, constructs flows of data that may be transformed while passing through different components, using their active proxies. These are similar to our channels. Our methodology is slightly different, in that we have leveraged the features of CORBA and its services and approach the problem from an operating system point of view, where the Ninja project takes a Java-centric Internet-service approach. We have also focused on the user interface, to allow simple data access and ease application development.
The Iceberg project [RKJ99] uses the Ninja framework [GWBC99, GWvB+01] to provide an infrastructure for integrated network communications. One of the main focuses is support for user device modality. Users can switch between devices and the system can adapt to the properties of the device. The Automatic Path Creation Service provides the mechanism to transform content on-the-fly [CMI, Pro]. This service is similar to our mechanism that provides dynamic data types by providing impedance matching of data types to build paths to convert data formats.

10.5 Data Interface and Distribution

Our API borrows concepts from the Standard Template Library (STL) [SL94, Gla97], which defines objects for organizing collections of data. The collections are typically located in the local address space, requiring the local node to parse data into components for insertion into the collection. The Java stream package, (Java.io) [Sunb], defines basic streams that may be adapted to add specific functionality. However, such adaptors may only be applied locally.

Several pervasive computing projects have investigated the problem of information access and sharing in heterogeneous environments. IBM’s TSpaces enhances the concept of a Tuplespace by adding consideration for heterogeneity of devices, scalability, and persistence [WMLF98]. TSpaces allow distributed applications to share information in a decoupled manner and allows a high degree of interoperability, via tuples. Their implementation includes support for access control, event notification, and efficient retrieval of information. In addition, new operators may be dynamically added to the server, which may be used immediately. This is similar to our design of allowing new container types to be spontaneously added to the system. The TSpaces project resembles a database system, where our system is more focused on adaptation of content. However, we could create a container type that was specifically tailored for tuples, which could be used as a shared data among applications.

The Infospheres project at Caltech is constructing an infrastructure for organizing task
forces [Cha96]. Their goal is to build a system that allows highly dynamic groups to be rapidly assembled and share information. Other concerns are how to scale to billions of objects, restricting access to objects to authorized personnel, dealing with message delays over networks that may scale globally, and managing resources by “freezing” and “thawing” objects when needed. This research is more focused on organizing dynamic groups of people.

The Information Bus is a mechanism for distributing data in large systems [OPSS93]. The system defines communication protocols and self-describing objects that encapsulate operations of specific types of data. The design distinguishes between two types of objects; service objects and data objects. Service objects represent a data type and contain the functionality to gain access to the data. Data objects represent the data and may constitute data from some source. In addition, the bus defines two methods of operation. Data can be retrieved either using a request/reply or a publish/subscribe model. New service objects can be dynamically added to the system at run-time to support new application needs, without interrupting the system. The system exhibits some of the same concepts as our system, such as wrapping data sources as objects to adapt different sources to present a more generic interface to applications.

The Document Object Model (DOM) is an object-oriented model to represent documents as a tree of nodes [CBNW]. Interfaces are available to traverse and manipulate the tree to gain access to structured data. The DOM interfaces are typically used as a result of parsing XML documents. Such documents can encompass an array of object types. We were more concerned with groups of similar objects, which simplifies our user interface. We could, however, create a container that resembles the functionality of DOM.
Chapter 11

Future Work

The *Gaia* file system evolved based on the requirements of applications in our active space environment. As new applications were developed and the infrastructure was used in new ways, changes and features were added to accommodate the changing set of requirements. Through its routine use, several areas for improvements have been identified. In this chapter, we discuss some of these improvements and some areas of future work that could be explored.

11.1 Dangling Pointers

A potential problem that exists in the system is the removal of a data source that has pointers to it, i.e., a file in a context-mode directory. For example, when metadata is attached to a file or directory, a link is created that points to the original data source. If the pointed to data source is removed, a dangling pointer will result; that is, there will be a pointer to data that does not exist anymore. To alleviate this problem, a reference count could be added to the data source. Only when the number of links referencing the data source is zero may it be removed from the system. This is analogous to the hard link mechanism on UNIX file systems. Our current mechanism is similar to shortcuts on Windows and symbolic links on UNIX, where dangling references are also a problem because they can span file systems and are not maintained in the kernel. An alternate approach would be to automatically garbage collect dangling pointers when they are found to be inaccessible.
11.2 Data Links

In the current implementation, metadata is attached to files and directories by creating a
directory that represents the collection of metadata tags and placing links to the real data
into the created directory. A mount point is then generated, which includes the metadata
and points to the created directory, resulting in two levels of indirection (see Fig. 4.14).
The reason for this design was to limit the amount of data that must reside in the mount
point so that they could more easily fit on a small mobile device. An alternate solution
would be to only maintain one level of indirection; the pointers to the source data would
reside directly in the mount point. This would increase the size of each context-aware
mount point, since the full path of all pointed to data would be included. For example,
suppose that the directory $C:\Temp\15687$ contains links to $C:\users\ckhess\ppt\cfs.ppt$ and
$C:\users\ckhess\ppt\gaia.ppt$. In the current design, the mount shown in Fig. 11.1 would be
generated if metadata of type $==$ ppt was attached to the files.

```
<CFS:Storage>
  <CFS:Owner>ckhess</CFS:Owner>
  <CFS:Host>srg181</CFS:Host>
  <CFS:Path>C:\Temp\15687</CFS:Path>
  <CFS:Context>
    <CFS:Type>type</CFS:Type>
    <CFS:Value>ppt</CFS:Value>
  </CFS:Context>
</CFS:Storage>
```

Figure 11.1: An example mount point using the current design.

If the second level of indirection was removed and the mount was to point directly to the
source data, the mount shown in Fig. 11.2 would result.

It can be seen that the size of the mount point increases as more links are added, which
is not the case in the current implementation. However, the removal of the second level of
indirection would simplify the design of the system and reduce the chance of corruption. The
<CFS:Storage>
  <CFS:Owner>ckhess</CFS:Owner>
  <CFS:Host>srg181</CFS:Host>
  <CFS:Path>C:\users\ckhess\ppt\cfs.ppt</CFS:Path>
  <CFS:Path>C:\users\ckhess\ppt\gaia.ppt</CFS:Path>
  <CFS:Context>
    <CFS:Type>type</CFS:Type>
    <CFS:Value>ppt</CFS:Value>
  </CFS:Context>
</CFS:Storage>

Figure 11.2: An example mount point using an alternate design.

trade off is between space and simplicity. Since mobile devices are possessing larger amounts of memory, the simple design may be a more desirable in the future as mobile devices evolve.

11.3 Attribute Advertisements

Attributes are tagged to data based on predefined specifications as to when certain contexts are valid and what types of properties applications are interested in. For example, our music jukebox application is interested in files of type \textit{mp3} and our attendance recorder runs when the \textit{situation} context is active. This has been manageable since we are dealing with a relatively limited set of applications and contexts, and an administrator of the system generally knows that kinds of data applications expect and when certain contexts become active. However, as the system is used for more tasks, it will become increasingly difficult to relay the expanding set of active contexts and application types to users and administrators of the system. Therefore, there needs to be a way in which applications can advertise (or be queried) as to what kinds of data they are interested in. In addition, the space must have some way in which to specify what context will be active during a certain situation or activity.

To make the possible contexts known, an ontology may be required that defines what
types of context information can be active in a space and in what situations they become active\textsuperscript{1}. Whenever a new type context can be detected, perhaps because a new type of sensor is added to the space, it should be added to the ontology. Space administrators could then inspect the ontology to determine the valid contexts that can occur within a space. Similarly, as new applications are installed in a space, they should register the kind of data they support. Our current methods have worked due to the limited scale, but new applications and uses of the space will necessitate a more formal specification of what and when context is available.

11.4 Native Interface

The current design of the file system is implemented at application level and includes an interface for applications. Since a new interface was devised, legacy applications cannot make use of the file system without changes. For example, we have developed a presentation application that uses PowerPoint to display slides. However, since this application must use the windows file system to access its data, we were required to wrap the application with a small software layer that retrieved and locally caches the file. A preferable method would be to support our features in a native file system. Several possibilities are available for implementing this: 1) build a custom file system device driver, or 2) implement the functions in a remote file server. The second approach is the simpler of the two because Windows file system driver development is complex. The open-source Samba project [Sam] would be a possible approach to accomplish this, due to its support for pluggable file systems. By creating a plugin for Samba and then remotely mounting the share, native applications could access our file system functionality directly. However, several features would not be supported and would have to be added separately, namely our mobile mount point mechanisms and data transformation support. However, mobile mount points are not directly used by applications,

\textsuperscript{1}This problem occurs in most context-sensitive systems.
so providing an out-of-band mechanism would not be problematic. Our data transformation feature could be wrapped on top of the native file system and could still use our new container interface. Since data transformation is typically not accessed by legacy applications, this would not pose a problem either, since applications that expect data transformation are written from scratch anyway.

11.5 Database Support

We implemented the mount server as a hand-coded database. In the future, it would be desirable to replace this with a commercial database on the back end. This would allow us to readily support a more powerful search syntax and would probably increase speed and stability.

11.6 Required Features

Our focus has been on introducing new concepts into the traditional file system to support applications in active spaces and therefore we have not implemented several features that would be required to make it a commercial-grade system. Below, we briefly list some issues that were not addressed in this work.

11.6.1 Fault Tolerance Policies

Ubiquitous computing environments can encompass a large number of computational devices and services. A key to the mainstream success of such systems will be in minimal/zero administration. That is, fault tolerance is critical for self-healing of systems and applications. Applications, however, may react differently to system failures. Furthermore, the fault policy may adapt to the context or device in use. This is generally true of all system services, not just file access. While this could be added to the file system, its proper placement
is in the low-level communications infrastructure. Applications could register, on a per-application/service basis, what the response to inaccessible servers should be. The choice of appropriate action depends on the application and what the user interaction is. Several possibilities are:

- **Retry indefinitely.** The application keeps trying until the service is again reachable (i.e., NFS [SGK+85]). This may be required for non-interactive applications.

- **Retry for a time period.** The application specifies how long to keep trying. This option could be used for retrieval of application data, but would give up after some time. The user could then try again later. This is similar to the current web browser approach.

- **Fail immediately.** The application would immediately abort further attempt to retrieve the data. For example, an MP3 player may skip any music files that it cannot retrieve and continue. In this case, the application performs a best effort approach and fetches any data that is available.

### 11.6.2 Reliability

If a file or mount server happens to fail, it must be automatically restarted so that clients can reconnect. This functionality should be available to all Gaia services. Several options are possible for implementing such functionality; 1) a monitoring service could be employed that detects when services fail and subsequently restarts them, or 2) services could be changed into Windows Services so that the Windows operating system performs the monitoring process.

### 11.6.3 Caching

Our system does not perform any caching above the buffer caching [MJ86] provide by the low-level operating system. File stores may be geographically disperse, since users may use servers hosted on a home computer from remote locations. Significant network delay for
reading and writing files can result for large network distances. Data staging is one method to increase performance of distributed file servers [GSSS02]. In this method, data that must travel large (network) distances are staged at nearby servers for quick access by clients.

### 11.6.4 Locks

The *Gaia* file system is for the most part a read-only system. Applications typically open files and use that data to display some information to the user. For example, when displaying presentations, the presentation data is created off-line and then made available to the space. In such cases, locks are not required. However, some applications do create data, such as the attendance recorder and calendar application, which would require the use of locks. However, *Gaia* is not typically used as a time-sharing system and normally multiple users are not accessing the same file for writing. Activities are defined by tasks and there is typically one task active at any time within an active space. Each application in the task deals with its own data. In addition, when multiple users do interact with the same data, for example in a collaborative application, access to data is serialized by the application model. Therefore, while locks are critical to ensure data consistency, the lack of a locking mechanism has not been a problem with the current applications that we have developed. The requirements of future applications may require that this functionality be added.

### 11.6.5 Consistency

Since *Gaia* is generally used as a read-only system, provisions for file consistency are not implemented. Other systems, such as the Andrew File System [How88], have addressed these issues and were not considered here. However, since there is no guarantee that two applications will not be writing to the same file at the same time, a mechanism for consistency is required. This could take the form of a callback mechanism and write-through.
Chapter 12

Conclusions

Ubiquitous computing is poised to revolutionize computer systems design and use. New challenges arise beyond what is required in typical distributed systems, such as how to integrate context, account for mobility, and integrate heterogeneous devices. Applications are no longer tied to a single machine, but may be mapped to a physical space based on resource availability or the role that the space is playing. Applications running in a physical space may be affected by context, such as the location, what is happening in the space, or who is present. Information becomes associated with the current task and the context may define the personal preferences or applications configurations. Our file system is used to address these issues by limiting the scope of information to what is meaningful for the current context, such as application configurations or application data, making personal information available conditioned on user presence, and adapting content for resource availability through dynamic data types. While much work has been conducted in location-aware data retrieval, we believe that other types of context information can be useful in such environments.

We have extended the concept of mount points to include meta-data tags to control the way in which data is organized in the directory hierarchy. Mounts are dynamic and can be injected into an environment to make personal data available to the local environment. The mount mechanism uses the tags to organize data so that data important to the current task is easy to locate, for a single user, group of users, or automated processes. The file system creates a virtual hierarchy where context is represented as directories and file system
primitives are used to manipulate the virtual hierarchy to attach context to particular files and directories.

The main contributions of the thesis are:

- Integration of environmental context into a file system design.
- Mechanisms to inject user data references into a local environment.
- A framework for supporting dynamic types.

A design philosophy of our work has been to create a programmable system in which the user retains a level of control. By allowing agreements between a space and its applications, users are able to tag data in such a way that it can become available to those applications conditioned on the current context. In this way, the system is able to operate with deterministic behavior. The applications that we have built are able to take advantage of the features of the system to simplify functions that usually require additional effort on the part of the developer or user.

The results of this work have simplified the construction of ubiquitous applications and have made making data available to space-hosted applications more convenient. Most importantly, the work has allowed us to explore the new ways in which ubiquitous computing spaces are used, how people interact with them, and the new types of features that must be considered in data access systems for ubiquitous computing.

This thesis has shown that these objectives were met though an evaluation of performance and applications which were able to leverage the new features that the file system introduced. While there remains many enhancements and improvements that could be made to the current system, the work presented lays the groundwork for future computing environments and suggests some novel ways in which such a system could be used.
References


142


Appendix A

The interface to the *Gaia* file system consists of two parts; the container interface to gain access to data sources, and an interface to query and manipulate the file system internals. In this section, we show these interfaces in C++ and briefly explain what each method is used for. For more details regarding parameters and exceptions, please see the documentation at http://choices.cs.uiuc.edu/gaia.

A.1 Container Interface

Specific container types are opened using a constructor for the class that represents the type. The *ObjectType* type in the interface descriptions refer to the data object type that a container holds and is specific to a particular container type. See Table 6.1 for a list of containers and the data objects they support.

Close the container and release all memory.

```c
void close ()
```

Set an attribute tuple. Attributes are sent with data access operations.

```c
void satt(const char *name, const char *value)
```

Get some objects from the container.

```c
int get(ObjectType &buf, int count)
```
Report if at end of container.

\textit{bool} \textit{eof}()

Put some objects in the container.

\textit{int} \textit{put} (\textit{ObjectType} \&\textit{buf}, \textit{int} \textit{count})

Test whether this container is valid.

\textit{bool} \textit{operator}!() \textit{const}

Test whether this container is valid.

\textit{bool} \textit{isValid}()

Tell the number of data objects in the container.

\textit{int} \textit{size}()

Set the current object index pointer.

\textit{void} \textit{seek} (\textit{unsigned long} \textit{position})

Tell what object index currently pointing to.

\textit{unsigned long} \textit{tell}()

Create a new generic container so that it can passed to another object and adapted to a type.

\textit{GaiaFS::Container} *\textit{open} (\textit{const char} *\textit{name}, \textit{const char} *\textit{kind}, \textit{int} \textit{mode}, \textit{int} \textit{index})
Duplicate a container reference.

\[ \text{GaiaFS}::\text{Container} \ast \text{duplicate} (\text{GaiaFS}::\text{Container} \ast \text{container}) \]

Add a listener of file system changes.

\[ \text{GaiaES}::\text{Channel} \ast \text{addListener} (\text{GaiaFS}::\text{Listener} \ast \text{listener}) \]

Remove a listener of file system changes.

\[ \text{void removeListener} (\text{GaiaFS}::\text{Listener} \ast \text{listener}) \]

Remove a listener of file system changes.

\[ \text{void removeListener} (\text{GaiaES}::\text{Channel} \ast \text{channel}) \]

A.2 File System Interface

The file system exports an interface beyond the container manipulation methods, such as changing the present working directory or checking the current mode the file system is in. Below is a complete list of available operations.

Check the accessibility of a file.

\[ \text{int access} (\text{const char} \ast \text{name}, \text{int} \ \text{mode}) \]

Set the current mode (file or context mode).

\[ \text{void mode} (\text{int} \ \text{mode}) \]

Get the current mode setting.

\[ \text{int mode} () \]
Refresh the mount table cache.

\textit{void refresh()}

Try to infer the mode from the path structure.

\textit{int mode(const char *path)}

Set the current context.

\textit{void context(Context &context)}

Gets the present working directory for the current mode.

\textit{const char *pwd()}

Return a clean version of the path.

\textit{char *path(const char *path)}

Set the current owner.

\textit{void owner(const char *owner)}

Get the current owner.

\textit{const char *owner()}

Get the home directory.

\textit{const char *home()}

Change current directory.

\textit{void chdir(const char *dir)}
Get the host that this manager runs on.

\texttt{const char *host(GaiaFS::FileServer *server)}

Lookup a path and get information to access it.

\texttt{Lookup *lookup(const char *name)}

Get the user portion of the directory.

\texttt{const char *user(const char *dir)}

Get the next active user name.

\texttt{const char *user(unsigned short &last)}

Get the next directory under this mount point.

\texttt{const char *mount(unsigned short &last)}

Tell if path is a mount point.

\texttt{bool ismount(const char *path)}

Add a new mount to the space.

\texttt{void mount(MountEntry &entry)}

Remove all mounts of the current user.

\texttt{void unmount()}

Print what is in the mount table.

\texttt{void print()}
Create a local version of the file.

\[ \text{const char *} \text{localize(const char *} \text{name}, \text{int index}) \]

### A.3 File Attributes, Modes, and Types

Data sources can be opened in different modes, depending on how the source is to be used.

- **Exist** Open a data source only if it exists.
- **New** Create a new data source; fail if it exists.
- **Write** Open a data source for writing.
- **Read** Open a data source for reading.
- **Append** Open a data source for writing and point to end of source.

The values corresponding to the two file system modes are given below.

- ** FileMode** File system is in file mode.
- **CtxtMode** File system is in context mode.

Data sources have a type depending on where the source exists. For example, a source with context meta-data attached may be a link to a file.

- **Dir** The source is a directory.
- **File** The source is a file.
- **DirLink** The source is a directory link.
- **FileLink** The source is a file link.
• **VirtDir** The source is a virtual directory.

• **CtxtDir** The source is a context directory.
Vita

Christopher K. Hess received his Bachelors and Masters degrees in Mechanical Engineering from Tufts University in Medford, MA in June 1991 and June 1993, respectfully. After leaving Tufts, he worked in New York City for 3 years before moving to Champaign, Illinois to begin his graduate work in the Computer Science Department of the University of Illinois at Urbana-Champaign. Christopher focused on multimedia protocols, networking, and applications and received his Masters degree in August 1998. He then shifted his focus to operating systems and ubiquitous computing during his PhD research. Christopher was one of the lead developers of the Gaia project, where he designed and implemented the Gaia file system.