Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '01)

San Antonio, Texas, USA
January 29 - February 2, 2001

An Adaptive Data Object Service for Pervasive Computing Environments*

Christopher K. Hess[1] Francisco Ballesteros[2] Roy H. Campbell[1]
M. Dennis Mickunas[1]

ckhess@cs.uiuc.edu, nemo@gsyc.escet.urjc.es, {roy, mickunas}@cs.uiuc.edu

[1]Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

[2]Rey Juan Carlos University of Madrid
Tulipan s/n. Mostoles
Madrid, Spain

Abstract

Workstations and PCs typically are rich in resources, in contrast to palmtop devices, which are generally quite limited. This disparity offers challenges to integrating these heterogeneous devices into a single distributed system. Services must be available to each device, but it may be necessary to modify certain services if the connected device does not have the desired resources.

A key component of many distributed systems is remote access to data. Traditional distributed file systems are typically rather static and are not able to adapt to the current available resources of the devices involved. Data files are treated as continuous streams of bytes and the interfaces to access them are designed for unstructured data; they simply transfer buffers of contiguous data. Providing modality and adapting content using these interfaces proves difficult.

In this paper, we present an adaptive data object service for pervasive computing environments using distributed objects. Data is manipulated through an object-oriented interface based on containers and iterators. The interface is also used to model data operations, conversions, and proxies. The system is aware of its environment and can instantiate objects in the proper locations to optimize performance.

*This research is supported by a grant from the National Science Foundation, NSF 98-70736.

1 Introduction

The recent popularity of personal digital assistants (PDAs) and Web-enabled cell phones has brought mobile handheld computing into the mainstream. Users are now able to perform many tasks that were once restricted to larger desktop systems. Although these devices will almost certainly always possess less computing power than their desktop counterparts, they will eventually offer universal access to the network. One of the key challenges is the integration of these handheld devices into larger distributed systems. The handheld devices should become an extension of the system that they can interact with in the same way that a stationary machine can.

The increasing diversity of devices accessing distributed systems makes traditional data distribution mechanisms inappropriate, since differing device types may require the service to behave in different ways. For example, when displaying video on a small device, it may be better to decode MPEG on a nearby host and send raw pixmaps to the handheld used for output. Systems that are not able to adapt to the current environment are therefore not best suited for heterogeneous distributed systems.

An active area of research involving highly heterogeneous environments has been that of pervasive computing [Wei93, Abo99, MIT, Hew, Mic]. These environments consist of intelligent rooms or areas, containing appliances (whiteboard, video projectors, etc), powerful stationary computers, and mobile wireless handheld devices. The large collection of devices, resources, and peripherals must be coordinated and access to them must be made simple. Such coordination may be viewed as being analogous to the role of a traditional operating system. However, the heterogeneity, mobility, and sheer number of devices makes the system vastly more complex [RC00]. Applications may have the choice of a number of input devices, such as mouse, pen, or finger; output devices, such as monitor, PDA screen, wall-mounted display, or speakers. An infrastructure for such a space must be able to locate the most appropriate device, detect when new devices are spontaneously added to the system, and adapt content when data formats are not compatible with output devices. For example, if a user wishes to view an on-going presentation on a small handheld, images of the slides could be sent to the roaming user, but in a format more appropriate for the device, such as a scaled down image to fit the small screen size. Moreover, more extreme transformations may be performed, such as converting text data to audio. 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 specific desired type. The system should automatically adapt content to the desired format and place the conversion modules in locations to maximize efficiency.

To address the foregoing issues, we have built a general data distribution service targeted at heterogeneous environments, that incorporates automatic content adaptation, location awareness, and knowledge of environment. The design of the service is based on the concept of containers and iterators exhibited in the Standard Template Library (STL) [SL94, MS96]; containers provide data manipulation operations, parsing mechanisms, and content transformations for structured data and convenient access is provided via iterators. Containers may be instantiated in the most appropriate locations, and access to these components may be transfered among nodes, enabling containers placed on various nodes to communicate. The application programming interface uses C++ templates and generic programming [Mus89] concepts to hide the communication infrastructure and maximize code reuse. In the current implementation, we have used CORBA as the underlying middleware layer. However, we are not restricted to using CORBA and are planning on porting the system to a light-weight communication core.

The remainder of this paper is presented as follows: section 2 gives an overview of our data service, including a brief description of the larger system the service is a part of. Section 3 describes the system layer of the service and the user layer containers and iterators, including examples. Section 4 presents our continuing work. Sections 5 and 6 present related work and concluding remarks, respectively.

2 The Data Object Service

The Data Object Service (*DOS*) is the data delivery mechanism for *Gaia*, an operating system for physical spaces we are currently developing. In the following sections, we describe *Gaia* and the design of the data service.

2.1 Overview of Gaia

Gaia is an infrastructure that exports and coordinates the resources contained in a physical space, thereby defining a generic computational environment [Gai00]. Gaia converts physical spaces and the ubiquitous computing devices they contain into a programmable computing system. *Gaia* is analogous to traditional computing systems; just as a computer is viewed as one object, composed of input/output devices, resources and peripherals, so is a physical space populated with many devices. An operating system for such a space must be able to coordinate the resources available in such a space. *Gaia* is similar to traditional operating systems by managing the tasks common to all applications built for physical spaces.

Gaia provides some core services, including events, entity presence (devices, users, and services), discovery, naming, location, trading. Devices are able to detect when they have entered new spaces and can take advantage of the services available in the physical location. By specifying well-defined interfaces to devices and services, applications may be built in a generic way that are able to run in arbitrary spaces. For example, a classroom application may be built that uses the physical devices in a room. When the user moves to a new classroom, the application can use the devices present in the new space.

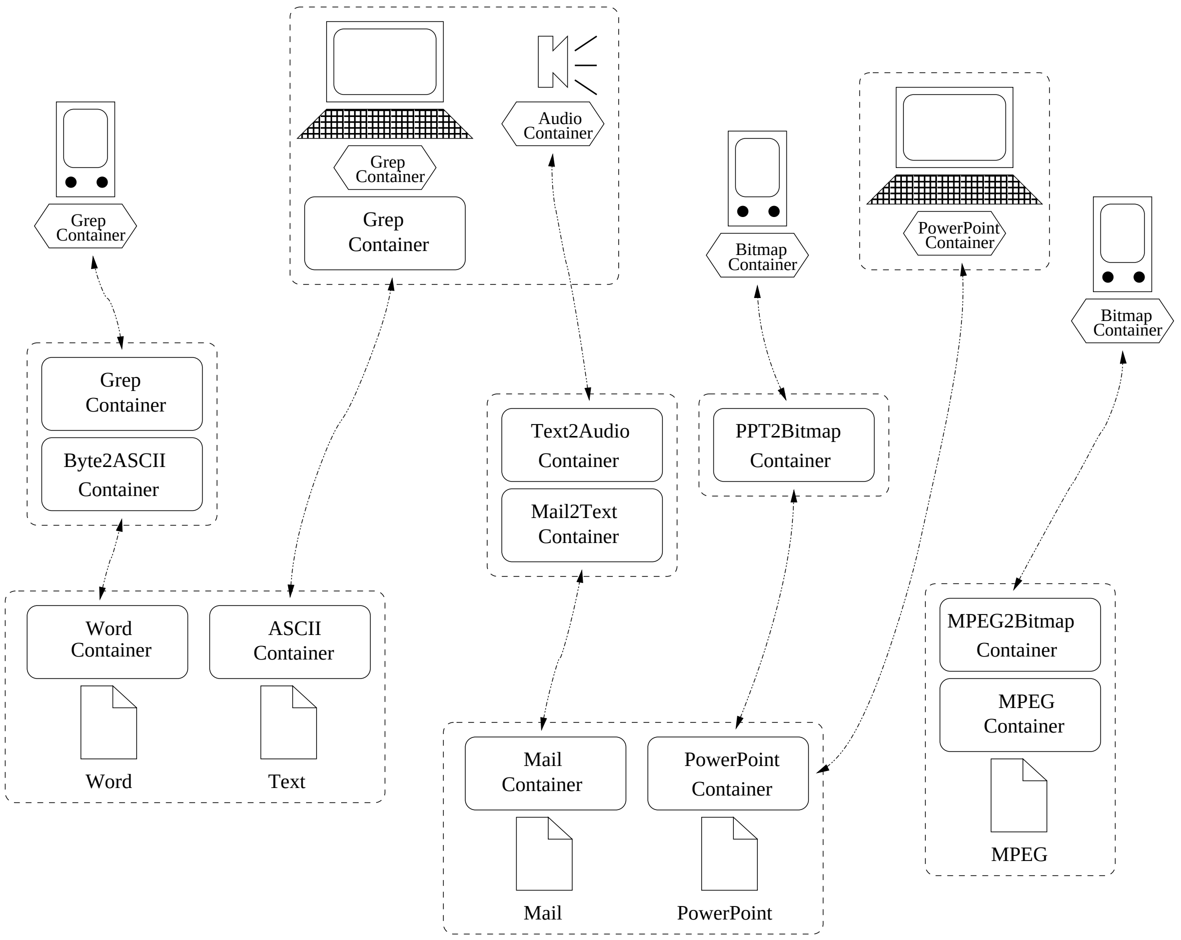

Figure 1: Servers manage their local native files and devices. The system can instantiate containers on any node in the system and adapt content for different device types. Rounded rectangles represent container instances. Hexagons represent template wrappers.

In addition, we are developing an application model for pervasive computing environments that is inspired from the Model-View-Controller (MVC) architecture. The MVC components are no longer restricted to software entities, i.e., they may be physical entities. For example, a house may be the model, providing data to various views of its sub-systems. Our model introduces an *adaptor* that can modify data from the model to a format the view desires.

Gaia is an extension of our previous work on the *2K* operating system. *2K* is a middleware operating system using CORBA [The98] as the communication mechanism and runs on top of existing platforms, such as Windows NT and Solaris. It uses a modified version of the TAO Object Request Broker (ORB) [SC99], called *dynamicTAO* [KRL+00], that offers dynamic configuration of the ORB internal engine in order to adapt to the dynamic needs of users.

Our initial implementation of *DOS* uses CORBA to leverage some of the standard CORBA services. In *Gaia*, we are applying these services in the context of physical spaces. Two services that are used heavily are the Name Service and the Trading Service. The Name Service allows transparent access to particular object references for applications. The Trading Service allows applications to find objects that possess some specific constraints. We are using these services in conjunction to provide location specific trading services. The Event Service is used with physical location detection systems (e.g., badges) to provide presence notifications of entities, such as users and devices. In addition, events are used to send "heartbeat" messages to the system to determine entity liveness.

Since *Gaia* targets pervasive computing environments, many small devices interact with the system. In the future, we will use a small composable communication mechanism, called the Universal Interoperable Core (UIC), that can communicate via different protocols (e.g., GIOP, SOAP) for mobile handheld devices that users may carry [UBI00]. The UIC can be composed dynamically, using only the required components. This allows the implementation to be customized to small devices and allows these devices to interact with services using standard protocols. In addition, devices can include server-side functionality, allowing them to accept events and method invocations. Since UIC is able to communicate with standard CORBA servers, we will be able to access the standard and custom services from these small handheld devices.

2.2 Data Objects in Gaia

Traditional distributed file systems [How88, SGK$^+$85, Wel92] are generally designed for homogeneous environments and simply transfer data to the local node. However, the heterogeneous nature of pervasive computing environments deems the static configurations of traditional distributed file systems inappropriate, since some nodes (e.g., handhelds) may require additional support from the infrastructure. Fixed policies may preclude some nodes from participating in these environments. A data access service that is dynamically configurable offers modality for different device types.

DOS is a middleware data service that makes use of the native operating system to manage data on disk. However, the service offers more than simple access to file data. In general, data is no longer transported as streams of bytes (although this mode is supported), but as data objects. Traditional file system interfaces (i.e., *open*, *read*, *write*, *close*) are replaced with object-oriented abstractions: *containers* and *iterators* [GHJV95, SL94]. These abstractions are a more suitable interface for accessing data as objects, since iterators can return data of a certain type and can be used to traverse the objects. Iterators provide the indirection needed to manipulate different containers using a single interface. In contrast to the standard *read* method for example, which passes a buffer to be filled in, an iterator returns references to objects whose size may be unknown *a priori* to the user.

In the most basic form, containers are simply wrappers for native file data or directories, but they can also be much more interesting and useful objects. In general, containers may represent *any* collection of data, that may be generated on-the-fly, gathered from disparate sources, or common data shared among distributed applications. They may also be used to interface with devices (e.g., writing a postscript file to a printer).

Containers are constructed as CORBA objects and applications can communicate with them through ORBs. CORBA provides infrastructure for transparent and platform-independent access to remote (or local) objects. Objects can be instantiated on any host and references to these objects can be passed around in a simple manner. This facilitates the creation of containers in various locations that may easily be connected together, as illustrated in Fig. 1. Dynamic placement of objects (and their functionality) is critical for heterogeneous environments to support all device capabilities. Different containers hold different kinds of data and CORBA handles the job of marshaling/unmarshaling and transporting the data. *DOS* assumes some of the burden generally placed on the programmer by parsing native file contents into indexed components that applications can manipulate more easily.

Some containers may be instantiated on *proxy servers*. These servers generally do not provide clients access to disk, but rather to their CPU and memory. For example, a proxy [Sha86] may be used to perform some expensive parsing or computation that should not be performed on the node maintaining the native files (as not to hinder other clients interacting with that server) or on the client (it is too weak to perform the parsing itself). The system can configure itself by placing container objects in the best location, based on knowledge of surrounding devices, to optimize performance.

For example, when performing a *grep* on a collection of files, simply copying data as-is to a small handheld device and searching locally may be inappropriate due to the severe resource limitations. It may be better to find matches on the file server and then transfer the resulting text to the handheld, as illustrated in Fig. 2. Clients can then use the search results to retrieve only those files that are of interest. However, the system should be configurable to direct where such operations should be

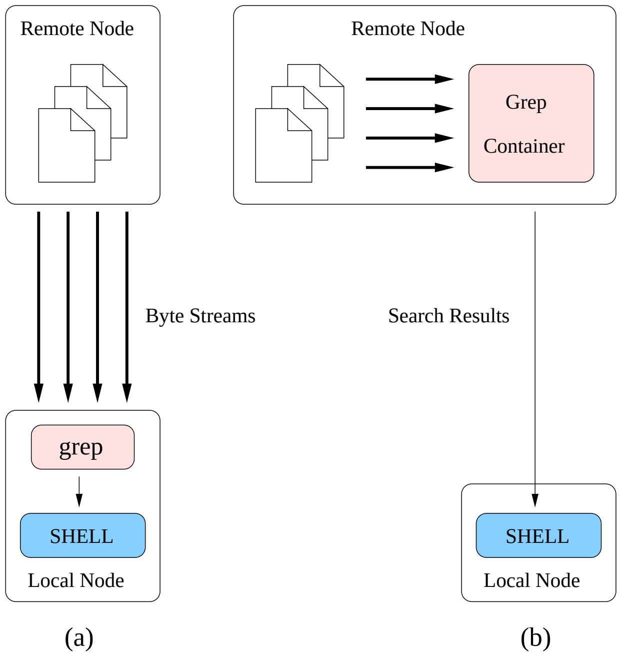

Figure 2: (a) Searching locally requires all data to be transfered to the local node. (b) Remote searching transfers only the results.

carried out to provide optimal performance. Powerful desktops connected with high-speed networks should not burden the server with these operations; they should use their own resources to complete the task. Weak devices should instead use the server computing power. Operations on data can be seen as containers that wrap a primitive data format and re-export it either as a different format or as the result of a transformation on the source.

As another example, Fig. 1 shows (on the right) a container translating MPEG to bitmaps for streaming video to a Palm Pilot device. The application can simply retrieve objects from an MPEG container and direct them to a display device, unaware of the complexity of establishing proxies and translating between data formats.

3 Architectural Design

The data service consists of two layers; a low-level system layer and a high-level user layer, as illustrated in Fig. 3. The lower layer has access to CORBA object references and includes a component to organize the storage naming hierarchy. The upper layer provides a simple user interface. We now describe both layers in the following sections.

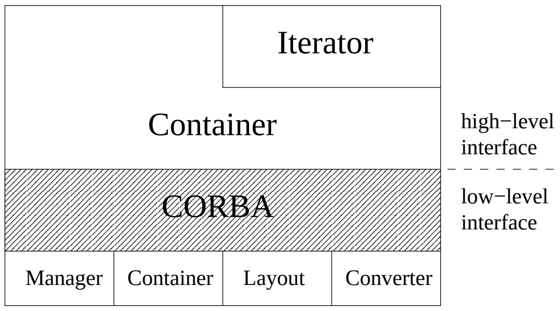

Figure 3: Layered structure of *DOS*.

3.1 User Interface Layer

The top layer consists of user level containers and iterators that hide the CORBA mechanics and reside in the local address space. Through a combination of wrapper classes and C++ templates, the user is presented with a clean and easy to use interface. Templates are used with generic programming concepts to provide a distributed generic programming model.

3.1.1 Containers

User level containers inherit from a template class that maintains a reference to the underlying CORBA container and provides methods for creation and adaptation. Container subclasses provide specialized operation for a particular container type (e.g., setting the dimensions of a slide presentation) and hide the existence of the template, providing a clean interface for application developers. If no special methods are necessary for a specific container type, the subclass is simply a wrapper and merely specifies the template parameter list. As shown in Fig. 4, the parameter list consists of the container type (C), buffer type (B), and object type (O), which are the CORBA container, transport buffer (a sequence of objects), and the indexed component object types, respectively. In addition, the subclass specifies the type of iterator to use. In this way, the application developer never sees the existence of a template, merely a particular container type (examples are given in section 3.2). The template glues together the correct combination of components for the container to work correctly. Containers generally do not need to add specialized code, so creating a container wrapper (only specifying the parameter types) can be done in one line of code.

Templates are used to provide compile-time polymorphism of CORBA container types, thereby applying generic programming techniques to distributed objects. Different CORBA containers provide methods to get and put objects of a particular type. However, the name of the methods must adhere to a convention (*getObjects()/putObjects()*) for each container. The particular object types to be transfered are specified in the template parameter list. Therefore, the template container transfers data of a certain type when communicating with the remote ORB.

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.

3.1.2 Iterators

Iterators provide a simple interface for users to traverse the structure of the data inside a container. They maintain the current position and cache information about their respective containers. Caching specific information about the container locally reduces the need to access the remote object as often, therefore, reducing network access and latency.

Different containers require different access methods and are associated with a specific iterator type. Since containers create iterator instances, the user is forced to use the correct iterator. The syntax for obtaining an iterator is identical to the STL and examples of its use are given below.

There are two types of iterators: *ObjectIterator*s and *StreamIterator*s. *ObjectIterator*s treat the contents of a container as objects, in contrast to *StreamIterator*s, that view the contents as a stream of octets. The latter are required to provide the traditional view of files, as streams of bytes, efficiently. The implementations of iterator types differ in how they detect when the iterator is at the end of a container. Subclasses provide specific methods for traversal. For example, *RandomObjectIterator* allows random placement of the iterator in the container.

Iterators are useful for retrieving remote objects incrementally [HV99] and containers hide caching of object groups. Although the container is a collection of items, the items need not all be loaded into local memory at the same time, as shown in Fig. 5. For example, when a user iterates over a collection of objects, they do not have to be individually pulled over from a remote server. Some number of objects may be prefetched and cached. The local template container plays the role of a buffer cache in standard file systems [MJ86]. If an object is requested, but is already available, it can be retrieved out of the cache. However, if the object is not available, the next group of objects may be retrieved to local memory and the current object passed to the user. The iterator hides this caching mechanism from the user; objects are handled as if they were all local.

The above described caching mechanism is used if data content is parsed remotely or on a proxy. However, if a container is resident locally, all data is transported to the local node and parsed there. Therefore, nodes with enough resources can cache the entire contents of a data source.

3.2 Interface Usage

The following examples illustrate the user interface. The first example opens a container as a stream of bytes in read-only mode. The container is then adapted to look like a container of text line objects. An iterator is then created and each line is printed to the console. Exception handling is removed for clarity.

```
ByteContainer b("MyFile", FS::Read);
LineContainer l = b.as("LineContainer");
LineContainer::iterator i;
for (i = l.begin(); i != l.end(); i++)
  cout << *i << endl;
```

It may be noted that although the containers are actually different template types, assignment is handled correctly. The *as()* method instantiates a CORBA *LineContainer* adaptor container on the local node (by default). However, the user or system may specify that it be instantiated remotely.

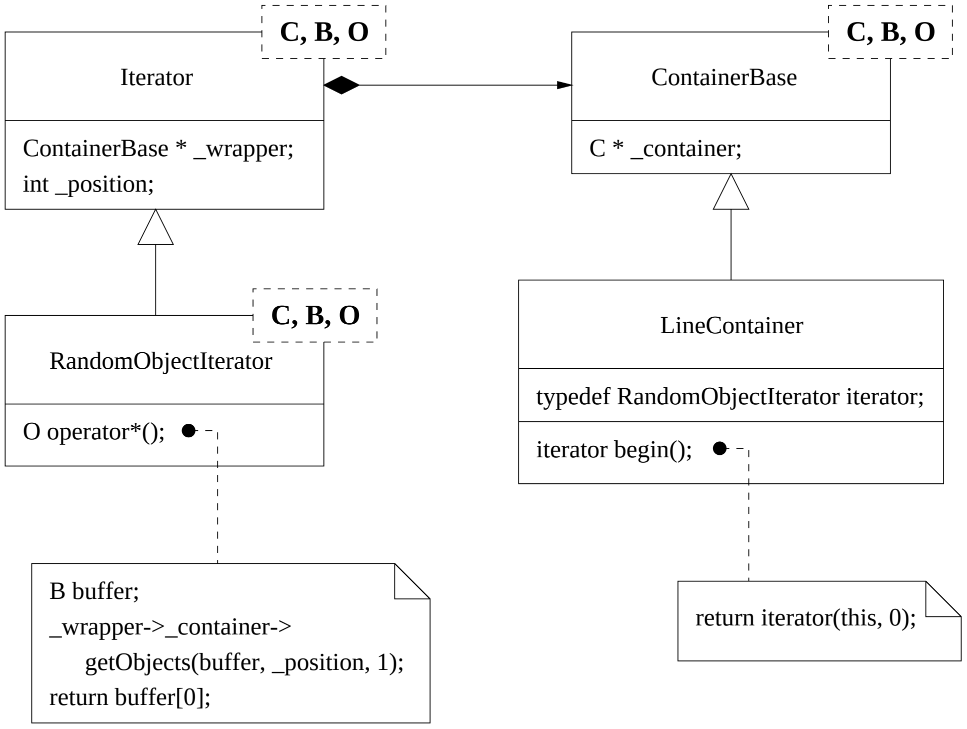

Figure 4: UML class diagram of relationship between *LineContainer* and *RandomObjectIterator* objects. Containers may retrieve groups of data objects and cache them (not shown) to reduce the number of network requests. **C** is *Container Type*; **B** is *Buffer Type*; **O** is *Object Type*.

Typically, the system uses the *as()* method to provide implicit adaptor instantiations; a container may be opened as a particular type directly, rather than having to first open it as a native container type and then specifying the adaptor. Therefore, applications can open containers in the format that they require and any adaptation/conversion is done automatically. This method is shown in the remaining examples.

The next example illustrates how a weak device may view a video sequence over a very slow network connections. Due to the limited resources available on such a device, it may be incapable of decoding and displaying MPEG video. However, the sequence may be transformed to bitmap images using a converter container and then pulled by the handheld device [HRCM00]. The container may need to handle the real-time nature of particular data sources, for example, by dropping frames if the client cannot keep up with the data source. It is the responsibility of the service to install the correct converter in the proper location, transparent to the application programmer.

```
BitmapViewer viewer;
BitmapContainer b("MyMPEG");
BitmapContainer::iterator i;
for (i = b.begin(); i != b.end(); i++)
  viewer.display(*i);
```

It may be desirable to "display" data in a format different from the source format when it is more convenient for the user. For instance, when using a computer with a small screen (e.g., a cellphone), retrieved messages may be more easily heard than read. A converter could be instantiated to present the data in the desired format.

```
AudioDevice device;
AudioContainer a("MyMailbox");
AudioContainer::iterator i = a.begin();
... get user input for message number ...
i += num;
device << *i;
```

The next example illustrates how a Palm Pilot can view a Microsoft *PowerPoint* presentation. The

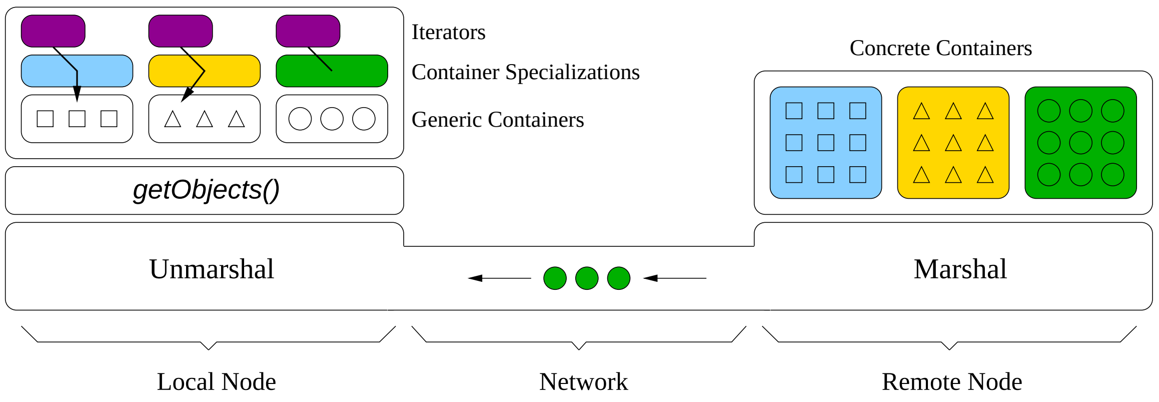

Figure 5: User-level containers are described using generic programming concepts to maximize core reuse. Typed objects are marshaled over the network. Groups of data objects can be cached in the local (generic) container.

system opens the presentation file with a *PowerPointContainer* (using OLE), which contains data objects (slides) in GIF format. It then converts the GIF slides to bitmap images using a *GIF2BitmapContainer*. The interface that the application manipulates is a *BitmapContainer*, which the *GIF2BitmapContainer* implements.

```
BitmapViewer viewer;
BitmapContainer p("MyPresentation.ppt");
BitmapContainer::iterator i;
for (i = p.begin(); i != p.end(); i++)
  ... get user input for next slide ...
  viewer.display(*i);
```

The previous examples implement different iterator types, but are used in a similar manner. The complexity of specific container and iterator creation are transparent to the user. Also notice that the containers create iterators to handle the specific data object types it holds.

3.3 System Layer

The system layer provides access to servers and servants via CORBA object references. A local component caches object references and provides name resolution support. Several types of system containers exist, which are hidden by the user level containers discussed above. The following sections describe the different types of containers available. In addition, the mechanisms that exist for locating data and creating components are discussed.

3.3.1 Containers

Containers are the main abstraction for representing data and provide methods for creation and deletion of the data objects they hold. Concrete containers are implemented using the *Gaia* component model. Each container is built as a dynamic link library (on Windows) or a shared object (on Solaris). The component model allows the service to load, create, and activate container components. Decoupling the containers from the service allows new container types to be added to the running system without interrupting current applications. There are several different container types that perform different roles in the system.

File Containers *File containers* enable access to native operating system files and directories. File containers parse data of different file types into indexed components (e.g., *DirectoryContainer*, *MailContainer*, etc). Parsing meta-data can be cached persistently for future container accesses, therefore eliminating the need to determine object boundaries each time a container is opened. This is particularly useful for containers that do not change frequently. Altering the contents of a container invalidates the cache.

There are several strategies that may be used when implementing a file container. Access mode and file size affect which strategy is employed. For example, if a container is accessed as read-only, the bytes on disk will not change. If parsing meta-data is available, access to indexed components only requires the server to seek to a component boundary (which is included in the parse meta-data), reading in the appropriate amount of data, and sending it to the client.[1] However, if the container is accessed read-write, the byte layout on disk will probably change. Information regarding the insertion and deletion of objects in the container are cached in memory and then committed once the container is released. Alternately, the entire container can be loaded into memory (i.e., as an STL container) and insertion and deletion of objects are performed in memory. When releasing the container, the entire contents of the in-memory container is written to disk. This strategy is implemented more easily, but requires more memory. An area of future work is determining when to use a particular strategy depending on access mode and file size. For example, large files should probably not be completely loaded into memory.

Some files do not contain any well-defined structure. Such files may be represented as a stream of bytes (*ByteContainer*), thereby supporting traditional file semantics. Finally, *ByteContainer*s can be used by applications that want to bypass the type system of *DOS*, be it for backward compatibility or due to the lack of an appropriate container type.

Processor Containers Containers can represent things other than standard files and directories. *Processor containers* act as "files" with dynamic content; the "file" is created on-the-fly. For example, a *GrepContainer* may provide the ability to perform remote *grep* processing on files in a directory. This allows the computation of pattern matching to be performed at a remote location and the results transfered to the client. This not only reduces computational overhead on a weak client, but network traffic as well.

Such remote processing may be performed on the server or at a proxy node. Too much processing on a server may slow down data access by other clients [SHG98]. If performed at a proxy, the server managing the native files acts as a traditional file server (just serving byte streams). Resource consumption is therefore split between two machines; the proxy server is used for memory and CPU, while the file server is used for disk access. This is managed quite easily through CORBA, since placing a component on a particular node is only a matter of directing a particular node to instantiate the object.

[1]More than one component may be sent in one request.

Converter Containers Weak devices may not be able to render data in its original format or process containers may require data to be in a particular formats [Wir]. Conversion of content is performed via a *converter container*, which is used to transcode data to a new format. Converter containers may be created on demand or automatically, when it is determined that the original data format is inappropriate, to provide on-the-fly transcodings.

Complex conversion may require the support of several converter containers; therefore, converters can be linked together. Converters can be created on different hosts, such as the local machine, the machine maintaining the native data, or any other machine. Creating an converter inserts a component into the flow of data and changes the container interface, similar to a module in a stream [Rit84].

For example, if a converter exists that transforms Microsoft *Word* documents into ASCII text format, a *grep* could be performed on *Word* files. Since *grep* requires ASCII text as input, the *Word* file would be opened as an *ASCIIContainer*, which the system would transparently convert in order to present the file in the format that *grep* expects.

3.3.2 System Core

The system core consists of a component that maintains a cache of references to machines exporting storage and provides name resolution facilities. The core includes a prefix table mechanism[2] [WO86] and, when needed, attempts to make connections to available remote data servers or proxy servers listed in the prefix table. Each remote data server manages the data content on their respective machines and is responsible for creating CORBA container objects on that host. A server may be started on the local node (if resources are available) so that the local disk may be accessed (if available) and containers can be created locally.[3] The interface to access local and remote objects is identical, so contacting any server is merely a matter of getting an

[2]Names are paths. Path prefixes, or "mount points", are translated to object references.

[3]Mobile handheld devices would probably not launch the local server and would rely on remote servers to instantiate all containers.

object reference to the correct server. This management component is a C++ class rather than a CORBA object, since it does not need to be accessed remotely.

The local view of the storage layout (namespace) is constructed through the use of the prefix tables. The prefix tables are used for name resolution and to locate storage. When a new file, directory, or device is accessed, the local container name in the hierarchy is translated to the native name and the manager finds the correct server hosting the content. Requests are then directed towards manager components (see section 3.3.4), which are responsible for the creation/destruction of various types of containers.

3.3.3 Layout Manager

The Layout Manager stores the prefix tables that allows machines and devices to export all or a portion of their storage. This manager is implemented as a service and may provide private local storage for a group or a physical space. For example, there may be a manager running in each space. When a user with a device enters a space, the device may obtain the storage descriptions of the space to build the local storage namespace. Another, more interesting, possibility is that the mobile device exports some of its storage. Consider a room that contains a projector and presentation software. The mobile device of the user may contain the actual presentation. When the user enters the room, the device contacts the Layout Manager and informs it of which part of its storage it wishes to export and the room then adds this storage to its namespace. The user may then navigate with the presentation software, which resides in the room, to the directory containing the presentation of the user, residing on the mobile. In such a scenario, there is no need to manually transfer files; the space automatically detects the existence of a new storage device and incorporates it. Hence, the namespace (i.e., what storage the room is aware of) can change dynamically as new machines and devices enter and leave physical spaces.

3.3.4 Container Manager

Access to each data source is initiated via a *Container Manager*. These managers act as factories for container creation and are the main entry point to gaining access to object references. Once a manager has successfully created an association between a container and a native file, processor, or converter container, a reference to the container is returned.

Container Managers also assist in data content adaptation/conversion, as described above, by finding an appropriate converter and returning a new interface.[4] Conversion may be done automatically by the manager when a request to open a container type does not match the underlying data source type. It may also be performed after a container has already been opened. This procedure is illustrated in Fig. 6. In order to adapt a container interface, a container object reference is transfered to the manager performing the adaption via the *adaptInterface()* method (Fig. 6-a). The manager determines the type of the container and examines a graph to see if a possible converter exists between the container type that was passed and the desired target container type. If a suitable converter container is found, a concrete instance is created on that node (each container provides a static *create()* method to generate an instance of itself on behalf of the Container Manager factory), and the new converter container is given the original object reference. The converter uses this object reference as its data source and knows the format of data that the source provides. Therefore, the converter receives objects of one type and sends objects of another type. The object reference of the newly created converter is then returned to the client, which can use it to get and put objects of the new type the adaptor supports (Fig. 6-b). Hence, containers can be linked together and the data can change as if flows through the links. Since these converters can be placed on various nodes, they may act as proxies for weak clients.

3.3.5 Container Descriptions

In order for containers to be linked together to provide the proper conversion, a description of the containers must be available. We describe containers using XML. 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, processor, or converter), input data object type, output data object type, and an optional file type (expressed as

[4]The new interface has the same method names, but handles different data object types.

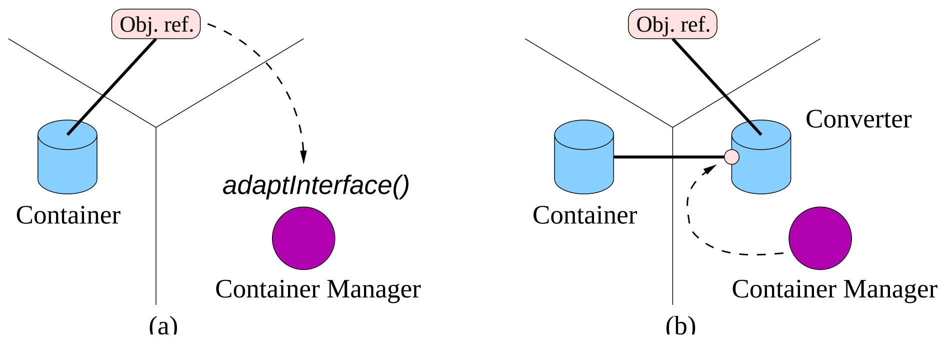

Figure 6: Container Managers enable adaptation of container interfaces.

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.

4 Continuing Work

Our current implementation is based on the CORBA standard. We will port our system to the UIC composable communication core to provide a light-weight implementation that can be used on small devices, such as Palm Pilot. In addition, the UIC can be composed to provide server-side functionality. We will develop a small server-side implementation that will allow a mobile device to automatically be added to the storage namespace of a physical space once it is detected by the space. This will allow scenarios, such as the one described in section 3.3.3, to be realized.

Another issue of future work is deciding the best location to instantiate containers (i.e., where to place proxy containers). We will use the *2K* resource manager [Yam00] for load balancing and to determine if a costly operation should be performed on a proxy node. Our service contains the mechanisms to place containers on various nodes, but this decision engine must be added.

Currently, client applications must pull objects. However, there are many situations where data should be pushed out to a client. For example, if a group of users are engaged in a discussion using a whiteboard, remote users may wish to see the schematics on the board. These updates to the contents of the board should be pushed out to remote users so that they can view new drawings. We will be adding push technology to our system to facilitate such scenarios by registering callbacks with containers. Real-time data may be streamed using RTP packets. We are building mechanisms to connect containers via *streams*, that will treat RTP packets as our data objects.

5 Related Work

Our work has a resemblance to file systems in some respects. Some previous systems have treated data as groups of data, rather that contiguous bytes of of unstructured storage. Semantic file systems [GJSJ91] index data when files and directories are created and updated. They allow extraction of attributes using file-type *transducers*. Such a system provides the user with alternate views of data and a query mechanism for finding information. The *Choices* file system [Mad92] defines a framework for building different file system types. Data on secondary storage is represented as containers and is parsed and indexed depending on file type. In addition, container contents can be viewed in different ways. However, the system is not distributed and does not perform adaptation and conversions. A replacement for standard file system organization has been proposed that logically treats files as nested boxes [BA99]. Remote copy operations and converters are incorporated into the design.

The effects of mobile code were evaluated on a distributed file service [SHG98]. The cost of performing remote file operations versus increase in server load was measured. It was found that moving operations to the file server (i.e., *agrep*) is typically advantageous when client CPU power is below that of the server and network latencies are high. However, excessive computational load on the server can reduce throughput for clients simply requesting byte streams. Our service can be configured to perform such computations on a remote node when appropriate.

Our API borrows concepts from the Standard Template Library (STL) [SL94, Gla97], which defines objects for organizing collections of data. It also defines generic iterator objects, similar to the C++ stream interface [Str98], to access the data of underlying data collections. Iterators form an abstract interface to a number of different collection types. 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*) [Sun], 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.

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. 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. Jini technology main focus is to allow devices to discover and interact with each other. Our system is more concerned with the delivery of data and adaptive content, rather than particular services.

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.

The work most similar to ours is that of the Ninja project from UC Berkeley [GWvB$^+$00]. The Ninja architecture defines four main components: bases, units, active proxies, and paths. Bases are manifested as a cluster of workstations that provide scalability, fault tolerance, and concurrency. Units comprise the myriad of devices that may be connected to the infrastructure. The active proxies provide adaptation of content (similar to our containers), and are the result of previous research in data distillation using the TACC [Fox] model to perform on-the-fly data transformations [FGBA96]. Transcoding data formats was found to greatly increase the performance of certain applications [FGG$^+$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.

6 Conclusions

DOS provides the data transfer mechanism in *Gaia*, an operating system for physical spaces. *DOS* is able to alter behavior based on knowledge of computing device characteristics and location. Data is represented by containers and access is gained using iterators. Modeling files and directories as containers unifies the interface for data distribution in our system; the interface is also used to model data operations, conversions, and proxies. Containers may be connected together as modules that can act on data passing through them. The system is aware of its environment and has the mechanisms to instantiate objects in the proper locations to optimize performance and provide load balancing.

We have hidden the details of CORBA from the developer by using C++ templates and wrapper classes and have applied generic programming concepts to distributed objects. Objects of a particular type are marshaled over the network and typecasting becomes unnecessary. User-level containers and iterators are defined as template classes that combine the proper components together to allow type-safe remote access to objects. Templates have made the construction of user-level containers trivial.

DOS is a dynamic flexible system, in contrast to typical distributed file systems, that are designed for a particular operating environment. The dynamic nature of our data service makes it well-suited for heterogeneous environments, prevalent in pervasive computing.

7 Acknowledgments

We would like to thank Fabio Kon and Manuel Román for helpful discussions regarding the design of our data service. We would also like to thank the anonymous reviewers for many valuable suggestions for improving this paper.

8 Availability

Resources for *DOS* and *Gaia* are available at:

`http://choices.cs.uiuc.edu/gaia`

References

[Abo99] G. D. Abowd. Classroom 2000: An experiment with the instrumentation of a living educational environment. *IBM Systems Journal*, 38(4), 1999.

[BA99] Francisco J. Ballesteros and Sergio Arevalo. The Box: A replacement for files. In *IEEE Hot Topics on Operating Systems (HotOS-VII)*, Rio Rico, AZ (USA), 1999.

[CBNW] Mike Champion, Steve Byrne, Gavin Nicol, and Lauren Wood. Document Object Model (Core) Level 1. `http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html`.

[Cha96] K. Mani Chandy. Caltech Infosperes Project Overview: Information Infrastructures for Task Forces. `http://www.infospheres.caltech.edu`, November 1996.

[FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to Network and Client Variation via On-Demand Dynamic Distillation. In *ASPLOS-VII*, Boston, MA, October 1996.

[FGG+98] Armando Fox, Ian Goldberg, Steven D. Gribble, David C. Lee, Anthony Polito, and Eric A. Brewer. Experience With Top Gun Wingman: A Proxy-Based Graphical Web Browser for the 3Com PalmPilot. In *IFIP International Conference on Distributed Systems Platformas and Open Distributed Processing*, Lake District, UK, September 1998.

[Fox] Armando Fox. The Case for TACC: Scalable Servers for Transformation, Aggregation, Caching, and

Customization. Qualifying Exam Proposal.

[Gai00] Gaia Research Team. Gaia: Enabling Active Spaces. `http://choices.cs.uiuc.edu/gaia`, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., 1995.

[GJSJ91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic File Systems. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 16–25, 1991.

[Gla97] Graham Glass. The Java Generic Libarary. *C++ Report*, 9(1):70–74, January 1997.

[GWvB+00] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Josheph, R. H. Katz, Z. M. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Special Issue of Computer Networks on Pervasive Computing*, 2000. (to appear).

[Hew] Hewlett Packard Company. Cooltown. `http://www.cooltown.hp.com`.

[How88] John H. Howard. An Overview of the Andrew File System. In *USENIX Winter Conference*, pages 23–26, Dallas, Texas, February 9-12 1988.

[HRCM00] Christopher K. Hess, David Raila, Roy H. Campbell, and Dennis Mickunas. Design and Performance of MPEG Streaming to Palmtop Computers. In *Multimedia Computing and Networking 2000 (MMCN00)*, San Jose, CA, January 25-27 2000. ACM.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley Longman, Inc., 1999.

[KRL+00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000.

[Mad92] Peter William Madany. *An Object-Oriented Framework for File Systems.* PhD thesis, University of Illinois at Urbana-Champaign, June 1992.

[Mic] Microsoft Corp. Easyliving. `http://www.research.microsoft.com/easyliving`.

[MIT] MIT Media Lab. Smart Rooms. `http://ali.www.media.mit.edu/vismod/demos/smartroom`.

[MJ86] Bach Maurice J. *The Design of the UNIX Operating System.* Prentice-Hall Software Series. Prentice Hall, 1986.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison Wesley Longman, Inc., 1996.

[Mus89] David R. Musser. Generic Programming. *Lecture Notes in Computer Science 358*, pages 13–25, 1989. Springer-Verlag.

[RC00] Manuel Roman and Roy H. Campbell. GAIA: Enabling Active Spaces. In *9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 17-20 2000.

[Rit84] Dennis M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(2):1897–1910, October 1984.

[SC99] Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine*, 1999. available at `http://www.cs.wustl.edu/~schmidt/ACE-papers.html`.

[SGK$^+$85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, Portland, Oregon, June 1985.

[Sha86] Marc Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *International Conference on Distributed Computing Systems (ICDCS'86)*, Cambridge, MA, May 19-23 1986.

[SHG98] Tammo Spalink, John H. Hartman, and Garth Gibson. The Effect of Mobile Code on File Service. Technical Report TR98-12, Department of Computer Science, Universiy of Arizona, November 1998.

[SL94] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical report, HPL-94-34, April 1994. revised July 7, 1995.

[Str98] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesely Longman, Inc., 3rd edition edition, 1998.

[Sun] Sun Microsystems, Inc. Java 2 Platform API Specification. `http://www.java.sun.com/products/jdk/1.2/docs/api`.

[The98] The Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification. `http://www.omg.org/library/c2index.html`, February 1998.

[UBI00] UBICore, LLC. UBICore web page. `http://www.ubi-core.com`, 2000.

[Wal] Jim Waldo. Jini Architecture Overview. `http://java.sun.com/products/jini/whitepapers`.

[Wei93] Mark Weiser. Some Computer Science Issues in Ubiquitous Computing. *CACM*, 36(7):74–84, 1993.

[Wel92] Brent Welch. A Comparison of the Sprite and Vnode File System Architectures. In *USENIX File System Workshop Proceedings*, pages 29–44, Ann Arbor, MI, May 21-22 1992.

[Wir] Wireless Application Protocol Forum, Ltd. Wireless Application Protocol Architecture Specification. `http://www.wapforum.org`.

[WMLF98] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Systems Journal*, August 1998.

[WO86] Brent B. Welsh and John K. Ousterhout. Prefix Tables: A Simple Mechanisms for Locating Files in a Distributed System. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, Cambridge, MA, 1986. IEEE Computer Society Press.

[Yam00] Tomonori Yamane. The Design and Implementation of the 2K Resource Management Service. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2000.