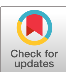

Extending Document Management Systems with User-Specific Active Properties

PAUL DOURISH, W. KEITH EDWARDS, ANTHONY LaMARCA, JOHN LAMPING, KARIN PETERSEN, MICHAEL SALISBURY, DOUGLAS B. TERRY, and JAMES THORNTON
Xerox Palo Alto Research Center

Document properties are a compelling infrastructure on which to develop document management applications. A property-based approach avoids many of the problems of traditional hierarchical storage mechanisms, reflects document organizations meaningful to user tasks, provides a means to integrate the perspectives of multiple individuals and groups, and does this all within a uniform interaction framework. Document properties can reflect not only categorizations of documents and document use, but also expressions of desired system activity, such as sharing criteria, replication management, and versioning. Augmenting property-based document management systems with active properties that carry executable code enables the provision of document-based services on a property infrastructure. The combination of document properties as a uniform mechanism for document management, and active properties as a way of delivering document services, represents a new paradigm for document management infrastructures. The Placeless Documents system is an experimental prototype developed to explore this new paradigm. It is based on the seamless integration of user-specific, active properties. We present the fundamental design approach, explore the challenges and opportunities it presents, and show how our architecture deals with them.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed databases*; D.4.3 [**Operating Systems**]: File Systems Management—*Distributed file systems*; E.5 [**Data**]: Files—*Organization/structure*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms: Design

Additional Key Words and Phrases: Document management systems, document services, user experience, active properties, component software

Authors' address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304; email: {dourish; kedwards; lamarca; lamping; petersen; salisbury; terry; jthornto}@parc.xerox.com.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2000 ACM 1046-8188/00/0400–0140 $5.00

1. INTRODUCTION

Whether managing email messages, spreadsheets, image files, or textual material, the world of the desktop computer is a world of documents, and most users organize their documents by location in hierarchies. File systems, for instance, impose a hierarchical structure of folders onto which users map their own semantic structures. More generally, hierarchies pervade document and information storage systems. Email systems group mail messages hierarchically, while Web browsers use hierarchies to store bookmarks. Hypertext systems have explored richer models for relating documents and document components, but have typically been focussed more on information retrieval than on document organization; and still, although a set of Web pages may be linked into complex hyperstructures, the HTML source is typically stored in hierarchically organized files.

Unfortunately, strict hierarchical structures can map poorly to user needs. The use of document locations as a fundamental organizing principle, and the restriction that documents appear in only one location at a time, forces users to create strict categorizations of document types and organization. Studies of actual document-filing practices suggest that these restrictions interfere with user needs. In the cases of real-world document collections, studies such as that of Trigg et al. [1999] or Bowker and Star [1994] show that categorization schemes are far less stable or absolute than they might seem. These problems carry over into the domain of electronic document management. For example, Barreau and Nardi [1995] present findings from a set of studies of file management on personal computers. They observe little use of deeply nested structure or cross-linking (via aliases or symbolic links), and instead note a preference for visual grouping and location-based search. Strict hierarchical filing can make it hard for users to do the following:

—*File documents:* Documents can appear in only one place in the hierarchy, even though they may play different roles and be relevant to multiple activities. For instance, a document concerning upcoming travel plans might be relevant to both budgetary decisions and to scheduling, but it can only be filed at one location in the hierarchy.

—*Manage documents:* Locations in the hierarchy conflate two roles, for document organization and document management. For example, not only are folders or directories used to group related files, but they are also typically the unit of administrative control for purposes such as backups and remote access. These administrative functions, then, impose constraints on the organization of documents. These demands make it harder to organize documents according to user needs.

—*Locate documents:* Documents may be filed according to one criterion, but retrieved according to another. However, hierarchical systems cannot represent the cross-cutting set of categorizations that might apply to a group of documents, requiring that document filing and document retrieval be performed uniformly.

—*Share documents:* An organization that makes sense to one person may not reflect the way that other relevant people think about the documents or need to group them.

To mitigate these problems, most file systems provide some "file pointer" mechanism such as aliases or links, allowing documents to logically appear in multiple folders. However, these can introduce as many problems as they solve. They create confusing distinctions among document name, document location, and document identity, which make them difficult to manage, by requiring users to understand the differences between acting on the link and acting on the target document. Moreover, these distinctions are managed differently in each type of file system; although they are intended to address similar problems, mechanisms such as aliases in MacOS, shortcuts in Windows, and hard and symbolic links in UNIX all have different semantics.

In the face of all these problems, we conclude that, while strict hierarchies of locations may provide a logical structure for document *storage* (meeting the needs of the system), they provide less useful support for document *interaction* (failing to meet the needs of users). This led us to investigate new models for user-centered document interaction.

1.1 Placeless Documents

This article reports on our ongoing research into a new approach toward document management. Our approach is based on document properties, rather than document locations. Properties are the primary, uniform means for organizing, grouping, managing, controlling, and retrieving documents. Document properties are features of a document that are meaningful to users, rather than to the system. Documents can have any number of properties, reflecting the different features that might be relevant to users at different times or in different contexts. To emphasize the difference between interaction managed through document properties and the more traditional approach of interaction managed through document locations, we call our system *Placeless Documents*.

The Placeless Documents design is based on three core features: uniform interaction, user-specific properties, and active properties.

Uniform Interaction. Most document storage and management systems already provide some sort of document property (or "metadata") mechanism. Even conventional file systems record details such as the owner of a file, its length, and when it was last accessed. Our approach takes this facility much further. In the Placeless Documents system, document properties are the primary tool for document management and interaction. They are used to record not only traditional metadata items, but also user categorizations, keywords, links to related items (such as earlier versions or procedure manuals), content-based features (such as indices or translations), and any other arbitrary value that a user or a program wants to associate with the document. This allows document properties to be used as a uniform interface for all document interactions.

User-Specific Properties. One important aspect of document properties is that a document has certain properties for a given individual. While one person might think of a document as a paper about Placeless Documents, someone else might think of it as an item in a clearance process, while a third person might regard it as a disclosure of intellectual property. In other words, document properties are expressed relative to the *consumer* of the document, rather than the *producer*. This is a fundamental aspect of the Placeless Documents design, and it manifests itself in two ways. First, the system emphasizes high-level properties of documents rather than low-level properties of files; and second, it supports the fact that different individuals may have different and independent sets of properties related to the same document. The property model provides the same interface for properties that are private to an individual or public to the world.

Active Properties. A system using properties as a uniform categorization model provides many advantages for user-centered document management. However, we can also use properties to *control document behavior*. In the Placeless Documents system, properties can be not only informative, or what we call static properties; they can also be effective, or what we call *active properties*. Like static properties, active properties can be assigned by users and added to documents; but in addition to this core behavior, active properties carry winnable code that can be invoked to control or augment document functionality. For example, a "backup" property can contain code that causes the document to be written to tape; a "summarize" property can cause a summary (text or thumbnails) to be generated whenever the document content is changed; and a "logged" property can cause all document accesses to be recorded.

Placeless Documents derives much of its power by combining these three features. By having both static and active properties, Placeless lets users employ the same uniform interaction model not only to manage and group their documents, but also to control document behavior and to extend and configure functionality. Active properties can give users control over the configuration and activity of system services such as content migration, backup, and load balancing. By combining active properties with a user-specific approach, the system can allow users to tailor the behavior of the system to match their own specific needs, while still sharing uniform access to document content.

This article presents the design of the Placeless Documents system. In particular, it focuses on the technical issues that arise in extending property-based document management with active properties, and in a system that uses document properties as a uniform interaction paradigm. The introduction of active properties into a document management system presents both opportunities and challenges; we describe how our architecture has developed to support active properties and how active properties influence application development.

In the next section, we outline the use of document properties as the basis for managing and interacting with documents, and discuss related

approaches. We then introduce the design of the Placeless Documents system before focussing in more detail on how we tackled the challenges of active properties.

2. MANAGING DOCUMENTS USING PROPERTIES

A variety of features make document properties an appealing model for the development of a document management system. Most significant, perhaps, for the needs of an interactive system is the uniformity of interaction that properties allow.

Document properties can be used to reflect features of the documents themselves ("size = 134k"); of the activities over the document ("last-read = Dec 15 8:22"); of the relationship of the document to user activities ("topic = budget"); and of the user's requirements of the document system ("backup.frequency = nightly"). All these different facets of document management can be managed through the same fundamental mechanism and hence the same interactive structures and styles. Properties like these can be used to group documents, to control their status and presentation, and to search for them; they can play the same role as file names, file system locations, user interface switches, and application-specific properties, but within a single, uniform framework.

A single, unified "property space" yields further benefits even with only static properties, since it also provides uniform integration between *applications*. Suppose the mail system records its information as document properties ("mail.from = lamarca@parc.xerox.com," "mail.subject = lunch meeting Wednesday"), and suppose that same property mechanism is also used by other applications such as the document summarizer, the access manager, word processor, contact database, etc. Then the features of each of these systems can be combined at a single level. Property-level searches, for instance, can refer to and combine the data elements of each application, which would otherwise require multiple applications with independent, private interfaces.

Since document properties are individuable entities, independent of each other, a property-based system can also manage the separate properties reflecting the perspectives of different users on the same document. Properties are an entirely compositional approach to document organization, so that multiple different views can be provided on the same document space.

We can also use properties to group documents. Placeless Documents offers a collection facility that allows users to group documents together and act on them as a unit. Membership in a collection can be derived dynamically according to document properties (although they also have static components, as will be discussed later).

Of course, not all "properties" are the same. Some properties are aspects of the document that are universally applicable to all document consumers, such as the document's creation time and the format of its content. In contrast, other properties are relevant only to specific individuals. When one user tags a document as "interesting," that should not affect others'

view of the document; it is a personal property indicating only their own relationship to the document. By the same token, other users might have attached other properties reflecting their own perspectives on the document; our user should not want to see those properties by default (although they should be able to ask to see them, if allowed). The property model provides natural support for these different aspects of properties. Since the model makes properties explicit entities in their own right, not merely derived features of documents, it allows us to make the property/document relationship one that is relevant to document consumers.

Active properties extend this uniform control to behavior as well as structure. Active properties encapsulate not only names and values but also active code. Their design is motivated by the simple observation that some properties have direct computational consequences. For instance, when Mark attaches the "important" property to documents, he might want that to mean that the document should be copied to a backing store regularly. Similarly, if Jonathan marks a document as "currently in progress," he might want an up-to-date copy to be maintained on his laptop for his next trip; and the fact that a document is being jointly authored with a colleague suggests that it should provide multiple different versions and be able to control and integrate changes that each contributor makes. In other words, a variety of properties expressing high-level user concerns can have consequences for how the system should operate. By allowing properties to carry executable code, responsible for performing relevant tasks that can achieve the needs expressed by these user concerns, we can use this same simple property mechanism to make the document management system active and responsive. In turn, using properties to control these features of system behavior allows a level of uniformity and flexibility that is hard to achieve in a world made up of tens or hundreds of separate applications and control panels.

The combination of document properties as a uniform means of document interaction and active properties as a mechanism for the delivery of document services constitutes a new paradigm for the development of document management infrastructures. Although some aspects of our approach have appeared in other systems, the uniform use of document properties as a means for users to manage and control document collections in a distributed environment results in a different interaction style and a new means of creating and delivering application services. The Placeless Documents project has been exploring the opportunities this new paradigm presents through an experimental infrastructure.

2.1 Related Work

Placeless Documents is not the first system to explore alternative models for document management, or the first to employ properties as a means to do this. Most file systems provide some model of properties or file metadata, although this has typically been fairly restricted. More modern systems, such as the file systems provided by BeOS [Giampaolo 1998] or

Windows 2000 [Richter and Cabrera 1998], extend these mechanisms to support arbitrary properties, but they do not use properties as the primary, uniform mechanism for document interaction.

The Lifestreams system, originally developed at Yale [Freeman and Fertig 1995], employs a “timeline” metaphor for managing personal document collections. All documents, including email messages, working files, etc., are organized according to the time that they entered the Lifestreams system. Filters can be applied to focus on particular documents, but the timeline is always the primary organizing metaphor for the document collection. Lifestreams shares some of our concerns with a uniform model of interaction, but differs in how this is realized by maintaining a primary, superordinate filing mechanism (the timeline) and in organizing documents primarily around system-derived properties rather than user-derived ones.

The Semantic File System (SFS), developed by Gifford et al. [1991], has some similarities to Placeless Documents in terms of its basic model. SFS uses a traditional file system interaction model, but backs it with a dynamic database rather than traditional file system storage. It provides “virtual directories” that are actually queries over the document collection, and provides arbitrary “transducers” that represent file system documents in the data tables. Collections in Placeless Documents share a number of basic features with the virtual directories of the Semantic File System, although, as will be discussed further, our collections provide additional features to aid in everyday interaction. Indeed, our focus on supporting direct user interaction for everyday document tasks is a primary point of departure from SFS, which was oriented largely toward being able to harness the power of a database in file-system-oriented command-line interfaces. Although Placeless Documents provides a means for file system-based interaction with queries and documents, we provide this largely as a convenience for the integration of legacy applications; in our model, “cd” and “ls” do not constitute a user interface.

Although a variety of document management systems provide some sort of document property facility, few provide support for activity. In Lotus Notes or Xerox’s DocuShare, applications can manage documents according to their properties, but the properties themselves cannot encapsulate active functionality. The Multivalent Documents work at Berkeley [Phelps and Wilensky 1996] can activate document content by means of small, dynamically loaded program objects, but provides control only over document content, rather than higher-level document management.

The notion of being able to attach code to documents as a means to control their behavior is similar to approaches taken in some other areas of systems development. For example, prototype-based object-oriented programming languages such as Self [Ungar and Smith 1987] or Cecil [Chambers 1992] adopt a model that is similar to ours; these languages focus directly on objects rather than classes and blur the distinction between slots (instance variables) and methods. In contrast to Placeless Documents, however, these systems are largely tools for programmers and provide little direct support for end-users.

Finally, issues surrounding the use of user-level code to extend system behavior have been explored in other areas. Operating systems, such as Spin [Bershad et al. 1995], have provided mechanisms for user-level extensions to operating system functionality. This allows them to achieve considerable performance improvement by allowing user code to run safely inside the kernel, both reducing the overhead of crossing the border between user space and system space and tuning the operating system policies to match user needs. Again, however, these facilities are provided at the programmer level, and correspond to a systems-level view of OS functionality, rather than being organized around high-level user needs.

2.2 Document Systems and Databases

One area of research that requires particular mention is database management systems. The basic relational database model reflects an approach that arose in response to the same set of concerns that motivate our work, by providing ways to express richer sets of relationships between items than strict hierarchies allow, and extracting structure as needed according to the needs of particular settings and situations. Object-oriented databases extend these ideas by combining data items with functionality to produce encapsulated objects, and defining relationships between them; and active databases also provide a model that allows activity and processing functionality to be added to data items [Kim 1990; Paton and Diaz 1999]. In many ways, these reflect a set of concerns similar to those that have shaped the Placeless Documents design.

However, other criteria have also influenced our design, and have led to significant differences from the traditional database approach. The first of these is that Placeless is a document system. This introduces a duality between metadata (properties) and content, and suggests that the system must have direct support for both. At the same time, it is crucial to our model that documents and metadata be maintained separately, so that we can dynamically incorporate document content from many different sources; so encoding document content as BLOBs (binary large objects) in a standard database would not be a sufficient solution.[1] The ability to incorporate pointers to external content within a framework based around metadata objects is not typically a core component of database systems (although systems such as Garlic [Cody et al. 1995] do provide support for encapsulated content objects). Second, Placeless is designed to support end-users directly. Through the use of properties and collections, users can create informal and fluid document organizations without predefined property taxonomies or schemas. Through the use of active properties, users have direct control over a compositional means for specifying system functionality. Support for this style of direct interaction requires simpler conceptual models for end-users, and sets different usage expectations and

[1]Similarly, this separation of metadata and content is also a distinction between Placeless Documents and document systems based on XML, although we can use XML to import and export documents.

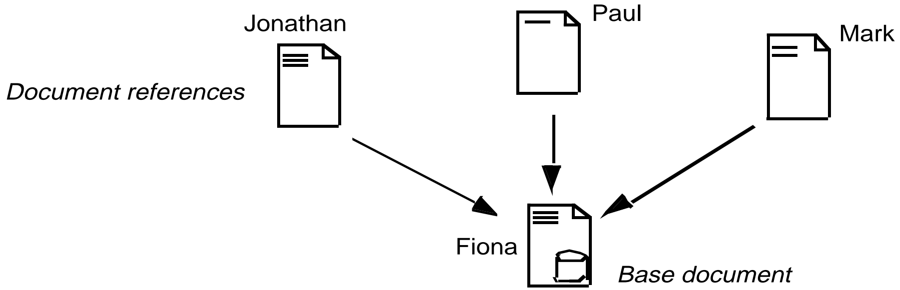

Fig. 1. Consumer-oriented properties. Universal properties are associated with the base document and can be seen by all users; personal properties on a document reference are seen only by holder of the reference, by default. So Jonathan sees six properties, while Paul sees four and Mark sees five.

effective optimizations. In particular, it leads to a different set of trade-offs for data structures and typing. The more fluid sorts of interaction in which end-users engage and the way that user information structures evolve over time mitigate against the use of rigid schemas and strong typing. Much of our implementation activity has been focussed on the consequences of this issue.

Of course, these are largely matters of emphasis. Database systems can, naturally, be used for fluid interaction and to manage external content, and indeed, there is a database at the core of the Placeless Documents system. However, the different patterns of use that an end-user document system encounters raises a different set of design and implementation strategies.

3. PLACELESS DOCUMENTS DESIGN

Before we can discuss the challenges and opportunities arising from the incorporation of active properties into document management systems, we need to set some context. This section presents the conceptual design of the Placeless Documents system and shows how active properties fit into the design of the full system. The design presented here is embodied in our current Placeless Documents implementation.

3.1 Personal and Universal Properties

Most documents that are of any interest have more readers than writers, or are interesting to multiple individuals. The different parties who interact with a document may have different opinions about the document, and may stand in different relationships to it. We call the users of the document "consumers," and our goal is to support consumer-oriented document management. Our document model, then, must accommodate the needs of multiple document consumers.

Placeless Documents supports two sorts of document objects, called *base documents* and *document references*, illustrated in Figure 1. A base document contains document content, while a document reference contains a pointer to a base document. Both base documents and document references

have a set of document properties; we say that the properties are "attached" to the document.

Document references combine their own properties with the properties attached to the base document. We call the properties attached to the base document *universal properties*, while those attached to a document reference are *personal properties*. For example, imagine that Jonathan holds a reference to a document written and owned by Fiona. Features that are universally true of the document, such as its format and its length, are universal properties; they are attached to the base document, held by Fiona, and so can be seen (access control allowing) by anyone who holds a reference to that document. On the other hand, Jonathan's annotations to the document are essentially personal assertions about the document; if he has marked it as being "interesting," that may not hold for other people. So, those properties are personal properties attached to Jonathan's document reference. They will not affect the view of other people holding references to Fiona's base document, unless someone explicitly asks, "what does Jonathan think about this?"

The owner of a base document can also hold a reference to that same document. This permits document owners to use the same separation of universal and personal properties that would be available to other users.

3.2 Static and Active Properties

Document properties have names and values. The property namespace is organized in a hierarchy for each document; this means that any property can use local subproperties for recording state information. For example, the "backup" property might use the subproperty "backup.lastrun" to record the time when it last ran.

A property value can be any serialized Java object.[2] So, although most user properties hold simple values such as strings, or slightly more complicated ones such as dates, application programs can use properties to record complex data structures.

In addition to their name and value, active properties include code that is executed when certain operations are performed on the documents to which they are attached. The operations that active properties can monitor include adding a document to a collection, adding or deleting properties, and reading the content. Active properties can intercept these operations before or after they occur, or contribute to the execution of the action, according to the needs of the particular application. By attaching active properties to a document, users can tailor and augment the default behaviors of the system.

3.3 Collections

In traditional systems, the logical structure of the document space is provided through the standard hierarchical structure of the file system.

[2]Placeless Documents is written entirely in Java. At the time of writing, it comprises approximately 100,000 lines of code written in Java 1.2, using JFC, JNDI, JDBC, and RMI.

The Placeless Documents design is an attempt to move away from that standard model, but it must still support the ability to group documents together. Our design allows users to group documents into *collections*.

The Placeless Documents system uses the "fluid-collection" design that we developed in an early prototype called "Presto" [Dourish et al. 1999a]. The fluid collection design balances the tension between wanting to be able to provide "live" collections backed by database queries and wanting to make these collections manipulable by users. In a purely dynamic design in which all collections were simply queries, manipulating the contents would be problematic, since modifications to the collection would be lost when the query was next evaluated.[3] Our goal was to be able to combine liveness with manipulability.

Fluid collections have three components. The first is a *query* over document properties; documents matching the query are members of the collection. The second is the *inclusion list*, which identifies specific documents to be included in the collection even if they do not match the query. The third component is the *exclusion list*, which identifies specific documents to be excluded from the collection even if they do match the query. Any of these components can be empty. For instance, if the inclusion and exclusion lists are empty, then the collection is defined entirely by the query. On the other hand, if the query is empty, then the collection has no dynamic component, and instead its membership is defined statically by the elements of the inclusion list.

The fluid collection design thus combines the benefits of dynamic and static collections. The query component allows collections to be defined dynamically, and their membership will grow and shrink as the document property space is manipulated. The two list components allow the results of the query to be modified directly, so that users can add and remove documents to the collection and have their modifications be stable over time.

Collections are, themselves, a form of document; so collections can be nested inside each other, can be assigned both active and static properties, can be organized according to the needs of individual users, and so on. In fact, in our current implementation, "collectionness" and "contentness" are two independent aspects of a document. Documents can have neither contentness nor collectionness, in which case they are "empty documents," essentially just collections of properties. With content features added, a document becomes a "content document," while with collection features added it becomes a "collection document."

Clearly, a fourth document type is available, that has both contentness and collectionness, implementing methods that access content and methods that allow their membership to be listed and changed. We call these

[3]One solution to this problem would be to let users manipulate the query indirectly by manipulating the content; however, investigations suggest that even forming valid queries is often a difficult task for end-users, and manipulating them indirectly would be doubly difficult [Greene et al. 1990].

"combined documents." Examples of combined documents in the real world are ZIP files, JAR files, mail messages with MIME attachments, and other formats that package and encapsulate other files. When these are implemented as combined documents, Placeless users and applications can access them as raw bit streams or as structured collections, as necessary.

3.4 Interfaces Above and Below

Although users see base documents as containing content, the Placeless Documents system does not, itself, store document content. Rather than being "yet another place to put your documents," the Placeless Documents system integrates and unifies existing document repositories through the use of what we call "content providers."

Every content document has a special active property called its content provider. This property knows how to read and write content from the underlying repository that holds it. For example, one class of content provider might know how to read and write to normal file system files on the host platform; another might know how to retrieve documents stored on the World Wide Web; and further content providers might retrieve the document content from an IMAP server or a custom database. The Placeless Documents system invokes the document's content provider whenever a request is made for the document content; the content provider contacts the relevant document repository and serves the document's content for Placeless. Some content providers may serve content that does not reside on any repository, but is instead generated dynamically, thereby supporting dynamic, virtual documents. Most documents, however, have "real" static content stored in an underlying repository.

The content provider mechanism provides a variety of benefits in the Placeless Documents system. One obvious benefit is that the system remains independent of whatever local document storage facilities are available. Another is that it applies our principle of uniform interaction across the range of document repositories already in everyday use. A third benefit is that it allows collections in Placeless Documents to be heterogeneous, containing documents that are actually stored in different repositories. A fourth is that content providers allow dynamic document content to be integrated seamlessly into the Placeless Documents framework. Finally, using content providers allows for easy extensibility to new document repositories or custom storage systems.

Content providers are a mechanism to get document content into the Placeless Documents system from underlying document repositories, but this is only half of the content path problem. The other half is offering document content, and Placeless Documents facilities are offered to applications that sit on top of the Placeless Documents infrastructure.

For developers writing new applications, the Placeless Documents system offers an API consisting of Java classes structured in terms of documents, collections, and properties. As discussed in the next section, applications may, in fact, be decomposed into a number of active properties, which are also written in Java.

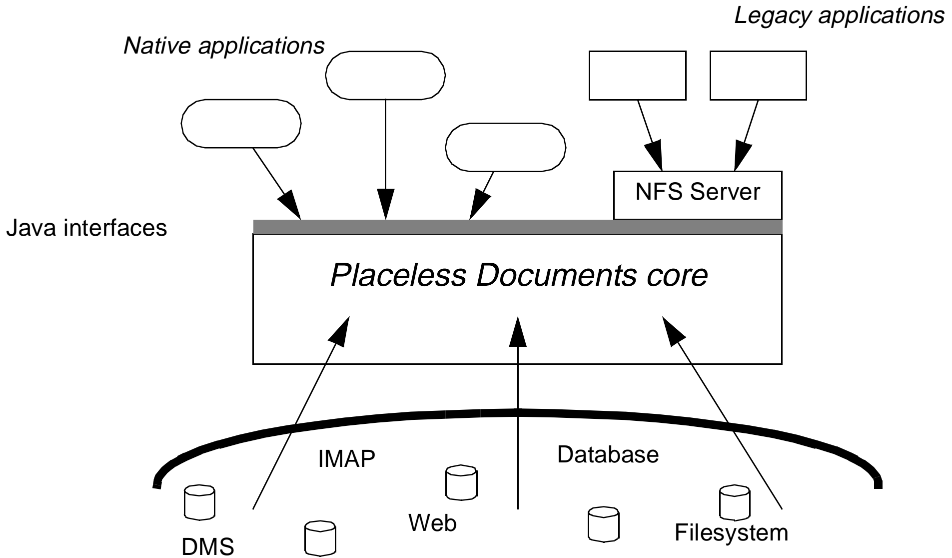

Fig. 2. The Placeless Documents system integrates content from multiple repositories and makes it available through a number of different interfaces. A native Java API supports Placeless-aware clients; the Java Streams interface supports Java Beans; and a custom NFS implementation supports legacy applications that expect to interact with a file system.

However, our goal is not simply to offer an infrastructure for the development of new applications but also to provide an integrated platform for interacting with existing documents, which implies interacting with existing document applications. Existing applications, of course, were not developed to use Placeless' document model; for the most part, they were designed to operate over normal file systems. So, to support these applications, we provide a file system layer on top of the Placeless Documents infrastructure. The Placeless Documents system implements an NFS (Network File System) server, providing access to stored documents through the standard NFS remote file access protocol [Sun Microsystems 1989]. This permits the Placeless Documents system to appear as part of the standard file system of a client computer, so that file-oriented applications can access documents stored in Placeless. Using this mechanism, standard "legacy" applications such as Microsoft Word or Adobe Photoshop can operate on Placeless Documents even though they know nothing about the Placeless Documents APIs and functionality. Of course, Placeless Documents and traditional file systems have different semantics and different approaches to document identity, which can cause some problems in trying to effect this integration. We discuss specific problems and solutions elsewhere [Dourish et al. 1999a].

The overall structure of the system is illustrated in Figure 2. The Placeless Documents infrastructure essentially acts as a distributed switch for document content, unifying disparate sources and making them seamlessly available within a single framework.

4. USING ACTIVE PROPERTIES

A principal focus of our research into Placeless Documents has been the opportunities offered by active properties. Active properties offer a means for users to configure, customize, and extend document system functionality using the same uniform interaction mechanism that they use for other document system interactions.

Active properties carry code with them. This code allows an active property to be involved in the execution of document operations on the documents to which the property is attached. Using active properties, then, users can control the behavior of the system on a document-by-document basis. The functionality of multiple properties can be combined on a single document. The interface signature of the active property code indicates to the system which events the code should be involved in. The Placeless core engine provides a property dispatch mechanism which invokes active property code during document operations, such as reading or writing content or adding properties.

4.1 Active Property Dispatch

Three sorts of active property code can be associated with each document operation, corresponding to three forms of involvement in the document operation itself. The dispatch engine processes each active property type separately.

When an operation is invoked on a document, the dispatcher first calls all relevant *verifier* properties for that operation. Verifier properties are intended to validate document operations; any verifier property can veto a document operation. Verifier operations can be used to perform fine-grained, document-specific access control.

If all the verifiers accept the operation, then the registered *performer* properties are called. Performer properties are those responsible for actually carrying out the requested document operation. Since different properties added to the document might affect an operation in different ways, performers have to be composed. For example, consider a document that has two properties, *compressed* and *encrypted*, each of which transforms document content. These properties register performer operations for the `readContent` operation, which constructs a stream for reading the document content. Their performer properties construct a stack of filter streams, each built on top of the previous one, each performing its own operation (compression/decompression, encryption/decryption) and combining to achieve the desired functionality. Performer sequencing is determined by a global order on document properties, which can be controlled by users.

Finally, once an operation has been performed, all relevant *notifier* properties are called. Notifier properties have no return value; they are intended for updating state, logging activity, and related functions. For example, a property that maintained an access log might register a notifier action for document operations; or a property that provided document

summarization might use a notifier to be informed when document content has been changed.

Naturally, an active property may be interested in more than a single event; the interface allows an active property to become involved in a variety of operations. For example, a property that maintains a history of writes to the document may want to be notified on calls to `addMember`, `writeContent`, `addProperty`, `setProperty`, and `deleteProperty`, since these all constitute "writing" operations of one sort or another. Active properties can implement any combination of verifiers, performers, and notifiers.

The separation into verifiers, performers, and notifiers simplifies the active property facility for both writing active properties and dispatching the operations. From the perspective of the active property writer, it allows programmers to focus specifically on the operations they want to perform. From the perspective of the dispatch mechanism, it simplifies property ordering issues by making a functional separation between different phases of property execution.

4.2 Extending Document Functionality

By associating new functionality with the core operations supported by the document system, active properties can specialize the behavior of documents in different situations. Active properties can also extend the behavior of the system to incorporate new functionality that is outside the core operations through a "delegation" mechanism.

One of the core operations that all documents support is `getDelegateFor`. The primary parameter to this method is an "interface" or abstract class signature, describing some Java functionality. The `getDelegateFor` operation returns a delegate object for the document that supports the named interface.

For example, suppose we want to extend the system to support language translation for textual documents. Although all our documents support the core operation `readContent`, there is no support in the core document interface for the methods `readFrenchContent` or `readGermanContent`. While language translation is not the sort of facility built directly into our infrastructure, we would like to be able to integrate it into our system by adding the "Translation" property to a document which extends that document's functionality with this new behavior.

With the delegation mechanism, we can achieve this by defining a new interface called `TranslatableDocument`, which contains the methods `readFrenchContent` and `readGermanContent`. When client programs want to be able to use the extended translation operations on a document, they call the `getDelegateFor` operation to ask for a delegate for the `TranslatableDocument` interface. The object that is returned from this call is one that directly supports the `TranslatableDocument` interface and acts as a translation delegate for the document. A call to `readFrenchContent` on the delegate operation returns the original document content

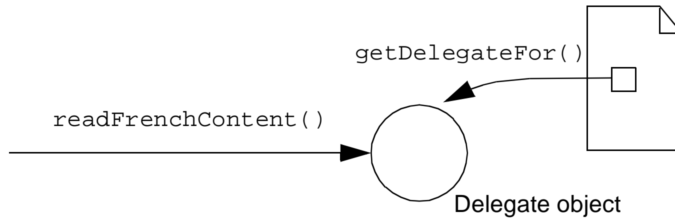

Fig. 3. The delegate mechanism allows an active property to extend a document's interface via a proxy object that delegates new methods for the document.

translated into French.[4] The relationship between the document, the active property, and the delegate object is illustrated in Figure 3.

The delegation mechanism permits applications to incorporate arbitrary functionality extensions into the Placeless Documents system. At the same time, it also retains type-safety. If we simply used a mechanism such as string identifiers to name and use document extensions, the compiler would not be able to ensure type agreement, and so run-time errors could result; at the same time, this would provide poor integration with the Java programming model that application developers would use. Instead, delegation lets programs deal with objects that directly support the extended operations as pure Java methods, so that the compiler can ensure type agreement in extension code.

4.3 Structuring Applications with Active Properties

Placeless Documents is a document management infrastructure, and so is intended to act as a substrate supporting a variety of applications. As outlined earlier, we provide a set of Java packages and classes allowing application developers to write applications that depend on the core functionality offered by the Placeless system. However, the active property functionality provided by the Placeless Documents infrastructure does not simply support applications, but transforms them. Active properties introduce a new form of application structure. Since properties are manifest directly at the programming level, they can also be created and manipulated in program code; active properties can be added to documents directly by applications.

Allowing applications to add active properties to documents essentially gives them the ability to delegate application functions to the documents themselves. The application functionality becomes distributed, in two senses. First, it can be associated with the documents themselves rather than concentrated in the application, so that it is distributed across the

---

[4]This raises important questions about document identity. Are translations in English and French really the same document? What happens when I make a copy the document—is my copy in English or French? Do references to other documents in the property values cause them to be copied too? Our current approach addresses these as issues to be resolved on an application-by-application basis, until we have developed more experience with the most effective mechanisms and those most natural to users.

whole system rather than a single server or application site. Second, these functions can be invoked as a consequence of independent action on the documents rather than action controlled by the application, so that they are carried out asynchronously, in direct response to document activity.

For example, consider an application to route forms through a business process. In the customary approach, the document-routing application would have to be running in order to manage this process; either users would operate within the system, filling out forms and indicating the completion of various tasks, or the application would be invoked when the tasks were completed in order to move to the next stage. In the Placeless Documents system, however, this can be achieved entirely with active properties [LaMarca et al. 1999]. The application is responsible for tagging appropriate documents with active properties, and then the properties will carry out the document-routing functions. For example, the routing application could operate by attaching a notifier active property for the `closeOutputStream` operation. This operation is called when the user finishes writing the document. At this point, the active property code can check that the document has been completed correctly. If it has not, then the user can be warned and asked to complete the form correctly. If the form is complete, then the active property can use the information it contains to route the form to the next relevant person. This routing application capitalizes on both features of the active property application structure. First, the application processing is directly connected to the documents involved, rather than the application that operates on those documents. In this case, the consequence is that any editor at all can be used to process the form; the user does not have to use the application's own form editor, and, indeed, users do not even have to use the same editor. Second, the activity of the application is directly connected to activity over the documents, making the application more responsive to user action and supporting the conceptual model of an "active" document.

5. SYSTEM ARCHITECTURE

So far, we have considered Placeless Documents on an abstract level. We introduced the motivations for our approach, and described our extensions to a simple static property model. This section discusses the system in more detail. In particular, it is concerned with two topics. The first is distribution and the question of how the functionality of the system can be distributed between multiple components to achieve reliability and responsiveness. The second is the kernel architecture and the question of how a document management system can be made fast and scalable in the presence of active properties.

5.1 Distribution Architecture

The Placeless Documents system is intended to be the primary means for users to store, retrieve, and interact with their documents. This introduces a range of important practical requirements. Placeless must be highly

reliable and available, even in the presence of network outages and overloads; it must scale well enough to support real environments, e.g., around 4000 hosts are registered in the domain `parc.xerox.com`; and it must operate in a world of firewalls, laptops, and modems, as well as in a world of highly connected desktop PCs.

To meet these demands, we designed Placeless around a flexible distributed architecture. This architecture supports a variety of distribution and replication schemes, but does not impose or require any; it provides a framework for mechanism but does not specify policy. This supports three goals. First, the design makes it easy for users and different implementations to participate in the Placeless system, without worrying about server configurations, replication policies, and so forth. The sorts of policies required for a large environment like PARC are unlikely to be the same as for running between a few machines at home. Second, the design supports using Placeless as a platform for the development of new distribution schemes. New schemes can be easily incorporated into the existing Placeless framework and will interoperate directly with existing client and server implementations. Third, the design enables us to tailor the distribution and replication policies to emergent patterns of document access. Since active properties are a new way of designing, implementing, and deploying document services, we cannot be sure in advance which policies will best suit this new environment, so our design is flexible enough to respond to patterns of real use.

The primary components around which the Placeless distribution model is organized are called *spaces* and *kernels*. A kernel is responsible for managing some set of documents, and provides the core document management functionality.[5] All operations on documents are performed in kernels. Together, one or more kernels serve a logical set of documents called a space (each kernel belongs to exactly one space). Spaces are associated with principals; typically, they correspond to individual users, although they can also correspond to groups. So, in addition to my own personal space, my project group might also have a space for documents that are not owned by any particular one of us.[6]

Spaces are given complete autonomy for the distribution of documents across their kernels. Different spaces, i.e., different implementations of the Space object, can provide different distribution policies within a single running Placeless system. The simplest Space implementation has only a single kernel that operates on documents directly. A more complex Space might maintain a number of different kernels, and divide documents between them, balancing the load of document operations across different

---

[5]So, the name "kernel" denotes a central role within our architecture; it does not denote any relationship to the operating system kernel. Our implementation is entirely in user space.

[6]Since document identifiers are unique across all spaces, the separation of personal and universal properties (Section 3.1) also operates across spaces. Since a user can have personal annotations on documents that are in a different space, organizing spaces by user is not strictly necessary from a conceptual point of view, although it is useful in managing protection domains.

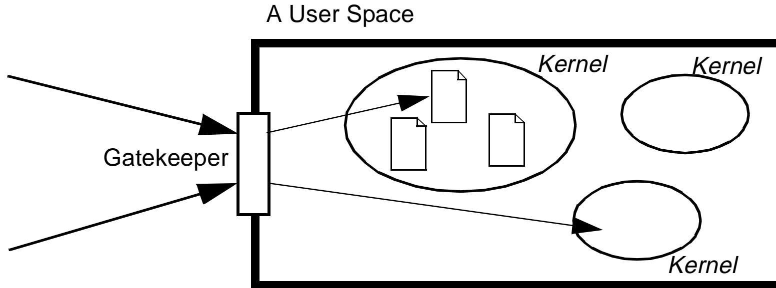

Fig. 4. Documents are managed by kernels, which are in turn organized within spaces. The gatekeeper is responsible for fielding document operation requests for a space and translating them into references to documents, implementing whatever distribution or replication policy is appropriate.

kernel processes (and, potentially, different hosts). A Space might also replicate documents across multiple kernels, to maintain high availability.

Spaces can also manage kernels that are not symmetric. For instance, a user might want one kernel on their laptop and another on their desktop machine. The desktop machine will be the primary document repository; but some documents should be maintained on my laptop too, so they will be available on trips away from the office. This Space might only copy documents to the laptop kernel on the basis of an explicit request (through a specific property), and maintain the desktop kernel as the default and master kernel for all document operations.

Although the document operations are actually performed by kernels, they are invoked on Space objects, which subsequently delegate to kernels. A gatekeeper is a Space object that is responsible for deciding which kernel will handle an individual document request, as shown in Figure 4. All document operations are performed in terms of abstract document identifiers, which name a space but not a kernel. When a document operation is performed, the gatekeeper for the document's space is contacted, and is given the document's identifier. The object that is returned is a direct channel to a document in a kernel; the space is, in most cases, not involved in further document operations.

Some operations, of course, must operate *across* kernels rather than being mapped onto a single kernel or document. Queries, for instance, may match documents stored in different kernels if the document space is distributed. Rather than delegating the search to a particular kernel, the gatekeeper is responsible for distributing search queries across all the kernels it manages and assembling the results.

Although a space is intended to manage documents across kernels autonomously, that does not imply that it receives no guidance from users. After all, a user is often better able to express needs and expected uses than a system is at guessing or extrapolating from current use. Expected patterns of use or current needs are, of course, simply properties of the document, and so can be expressed by attaching document properties. For example, a property that monitors all writes and updates to a document

can ensure that an up-to-date copy is maintained on a specific kernel, such as on a laptop. In similar ways, active properties can ensure that a document is always maintained on a high-availability server, should be replicated, must not be replicated, and so on. In other words, properties can be used to control the distribution policy and hence to achieve different levels of service. This is not the first use of document properties to control quality of service (see, for example, the work of Borowsky et al. [1997]), but it offers this level of control within the same uniform framework as other aspects of document management.

5.1.1 *Caching*. In a fully distributed implementation, each component of the system (space objects, kernels, documents, etc.) can potentially be remote to an application, residing on a different host or in a different address space. The use of public-key certificates and other security tokens means that the arguments to the remote calls may be large. Since we operate in an environment that includes laptops and mobile devices, components may become disconnected from the network. In the face of these problems, we still want our system to be responsive and stable.

We introduce a multilevel caching approach for all potentially remote objects [de Lara et al. 1998]. The core components of the system, such as documents, are defined not as classes but as interfaces that objects may implement. This allows multiple different implementations to be used seamlessly. Any given object may be implemented by a direct local object or by some form of proxy, which can stand for a remote object without any changes to an application. Using proxies, we can allow a wide variety of local/remote configurations, but the other problems still remain.

Allowing proxies to perform various levels of caching can overcome these problems. Remote caching can be achieved by introducing proxy peers: server-side objects that correspond to a client-side proxy. For instance, consider an application making use of a document object. If the document object is local, then it might hold the document object directly; but if the document is remote, then it might instead hold a document proxy. The document proxy implements the standard document interface, so that it is indistinguishable to the application, but delegates these operations to a remote object on the server side, which is the proxy peer for this particular document proxy. Since that proxy peer corresponds only to this particular *instance* of the document, the proxy and proxy peer can cache information to reduce network traffic, such as the credentials of document operations.

Using document proxies can also help make the system more stable. Consider the case where an application holds a document proxy object, which corresponds to a document on a particular kernel, in this case, a kernel on a laptop. When the laptop is disconnected, the document would normally become unavailable to applications using it. However, in the presence of replication, the application's operation could conceivably be completed against another copy of the document. To support this, our proxies can also *rebind* a document pointer to a kernel document object by detecting the disconnection of the original kernel and going back to the

gatekeeper to request the document a second time. Rebinding allows applications to operate robustly in a fluid environment.

In summary, our caching architecture supports a range of effects to improve performance and robustness and reduce network traffic and client/server communication.

5.2 Kernel Architecture

The Placeless Documents kernel is the basic component responsible for managing a set of documents, recording their properties, performing basic document operations, and responding to queries. The kernel needs to be both fast and scalable (although our distribution architecture, described above, also provides for scalability).

With only static properties, an effective implementation is reasonably straightforward. Our first prototype system provided no support for active properties, and so could adopt a fairly simple kernel design, keeping document objects in memory along with an index designed for fast query performance [Dourish et al. 1999a]. However, the inclusion of active properties introduces a number of problems into the design of the kernel. Our prototype solution was not appropriate for Placeless, which requires greater scalability than an in-core approach would allow.

The current Placeless Documents kernel, then, uses a relational database as the primary live metadata store.[7] The kernel maintains a fixed size in-core cache of document objects for faster response, but the database stores the "true" copy.

This design should allow queries over a kernel's documents to be transformed into database queries in SQL and performed directly in the database. However, the presence of active properties can introduce a problem. Our design made retrieving a property's value one of the basic document operations that active properties can override. This opens up the possibility of properties with dynamic values, calculated on-the-fly as they are read. For dynamic values, the database's index is useless. Testing the property value involves running real code, and that code can only be run in the Placeless kernel, not in the database.

When we first encountered this problem, our solution was what we call the "cache/refine kernel." The cache/refine kernel distributes the work of the query between the relational database and the kernel itself, attempting to exploit the areas where each is powerful. The basis of the cache/refine approach is that, although we cannot always be sure when we store a property in the database what its value will be when we read it next, we do know two things. The first is that we know there is a property with that name, since properties cannot change their names. The second is that we can know whether or not that property has a static value, because we can tell the difference between active and static properties. The combination of these features allows us to distribute the work of evaluating a query. When

[7]Placeless uses the JDBC interface to access a variety of databases; we currently run our implementations against Oracle and MySQL.

a property is stored in the database, it is marked to record whether or not its value is potentially dynamic. Later, when a query is processed, it is transformed into SQL and handed to the database. However, the query is generalized at this point; instead of asking, "tell me all documents where property 'important' is equal to 'true,'" the database is asked, "tell me all documents where property 'important' is *possibly* equal to 'true.'" This set includes those documents in which the value of property "important" is statically defined with value "true," along with those documents that have a property called "important" with a dynamic value. Once this generalized set has been returned from the database, the kernel itself is responsible for testing the documents with dynamic property values and eliminating those which should not be returned from the query. In this way, we can exploit the scalability and fast index-based lookup provided by the database as well as the dynamic values provided by kernel-based active properties.

As our experience with the system grew, however, we came to reconsider particular aspects of the design such as the active properties that could compute their own values dynamically. Initial application experience suggested that this feature was only occasionally useful, but was always costly. The current Placeless Documents implementation foregoes the cache/refine kernel and relies on the delegate mechanism for applications requiring dynamic property values, enabling more aggressive caching for faster queries. The issues of dynamic data and the distribution of indexing and caching functionality across kernel and database are ones that our ongoing implementation efforts continue to explore.

5.3 The Placeless Object

Kernels, spaces, and proxies are features of the architecture but do not form part of the basic programming model. To act as a unifying point of entry into the system, we introduce an entity we call "the Placeless object." This object acts as a single point of contact for an application or for a number of applications sharing an address space, similar to Gamma et al.'s [1995] "Facade" pattern. The Placeless object is responsible for contacting spaces, finding gatekeepers, managing credentials, and various other functions that are necessary for architectural coherence but need not form part of the programming model at the top level.

5.4 Security Model

The combination of active properties, document references, and a distributed implementation unavoidably implies that document operations may cause user code to be run dynamically on a variety of possible hosts, including user workstations and servers. In order to deal with the issues that arise in this scenario, our architecture has been designed to support a strong security model. As a purely practical concern, a strong security model is an absolute necessity before anyone can be prepared to trust their documents or their workstations to the Placeless Documents system.

Fig. 5. Reducing network traffic by using credentialled objects. Instead of passing secure credentials on every call, users can obtain a unique proxy object, which wraps their credentials on the server side.

Placeless' security model is based on *secure credentials*. At the kernel level, all API calls require credentials as an argument. Credentials identify the user under whose authorization the operation is being carried out. The kernel uses credentials to ensure that a user has authority to perform the given operation, either because the owner of the document has authorized the operation, or because some other user (or chain of users), authorized to delegate permission for the operation, has done so under some currently valid set of conditions.

In order for this scheme to be effective, credentials must be unforgeable. We can achieve this by using public-key certificates as credentials. Public-key certificates are digitally signed and authorized, so that they can be trusted as secure identifications of the originating user. However, public-key certificates can be large, and if they are used in a capability-passing style, a single operation might involve multiple certificates; they would quickly dwarf actual content on network transfers. To address this, we exploit the caching model described earlier. Rather than having to pass the credentials over the network for every single document operation, users can present their credentials once to create a server-side proxy for the document object which encodes their credentials. The server returns a unique handle for the credentialled object to the originating user. Further operations on this document handle need not pass credentials across the network; the server-side proxy will introduce the credentials into the local call in the kernel (as shown in Figure 5).

6. DELIVERING DOCUMENT SERVICES WITH ACTIVE PROPERTIES

The preceding sections have introduced the basic conceptual model of the Placeless Documents system, and explored the architectural implications of active properties as a means for document management. Now that we have seen how active properties can be supported in a system design, we focus on how they can be used in document applications.

One important role active properties can perform is the delivery of document services. Just as active properties result in a new model for application structure, they can also transform document services. This transformation takes place in two ways.

First, property-based document services are centered on the document and document activity, rather than on a separate application. To invoke a document service such as translation, summarization, or format conversion in a traditional model, users must either download their document to a network-based service, or fire up the relevant application to provide that service. In the Placeless Documents approach, they can operate directly on the document; the properties to provide the desired service are attached to the document and are invoked by operations over the document itself.

Second, document services are "componentized." Individual properties can be attached to individual documents, operated on individually, and composed directly at the site of activity. The active property mechanism sets up a framework for properties to interact with each other and "mix and match" coherently under user control.

6.1 Content Transformation

One sort of document service that can be provided easily through an active property is content transformation. Content transformation can take different forms. For example, automatic language translation of textual material could be one form of content transformation; or automatic annotation of document content with information about the document's history might be another. Content transformation can also affect the basic storage format; documents might be converted between the formats supported by different word processors, for example, or between different image formats. Finally, content transformation also includes those transformations that add value to the document; for example, an encryption/decryption filter can be implemented as a form of conditional content transformation (from an encrypted form on disk to a decrypted form on the user's screen).

We discussed earlier how content transformation could be achieved using the delegation mechanism. However, this mechanism would only work with applications that understood the specific semantics of the delegated operations, and knew to call them. We can also integrate content transformation directly into the normal reading and writing of documents.

This sort of content transformation can be achieved easily with active properties, since active properties can get involved in the construction of input and output streams. The Java I/O model allows streams to be "layered," in which different classes can perform "filtering" operations on the content to be read or written. By carrying code to take part in the creation of an input stream, for example, an active property can insert a filter that transforms the document content before it reaches the user. This filter could be used for content transformation such as language translation. The code that the active property carries will be called directly as a result of read operations, at which point it can send the content to a network-based service or perform the service itself.

6.2 Access Control and Digital Property Rights

The use of verifier properties, which are offered the opportunity to veto specific operations, provides an opportunity to perform creative forms of access control.

All operations are validated against the credentials of the user invoking the operation; since credentials can be passed around between users, Placeless Documents can support a "capability" model in which rights are first-class objects in the system. This mechanism is the basis of the system's security model and ensures that identities and access rights can be reliably and accurately validated.

However, since arbitrary code can be encapsulated in an active property and executed as part of the operation verification process, more complicated solutions are also possible. For instance, suppose Jonathan has a document that he wishes to make widely available for the payment of a small fee. So that people can find out whether they are interested, he might allow them access to a small summary of the document but restrict access to the full content until they have made a payment.

He can achieve this with active properties by writing a verifier property that will contact his server and check the credentials of the calling user to see whether or not they have paid and whether or not they have been validated for access to the document. When this property is attached to the summary, it can be handed around and checked by people, but each read operation will be dynamically validated.

However, a more interesting solution comes from taking a leaf out of the "content transformation" book. Jonathan's active property could *transform* the content depending on whether or not the user has paid the fee for reading the document. If the fee has been paid, then the user will receive the full content; but if not, the content will be replaced with the summary and an advertisement for the full document. This can be managed because Jonathan's code is called every time a document input stream is created. In fact, there are other advantages too. Since the property is attached to the document, it also travels around with the document when the document is copied, emailed, etc. So the document can securely pass from one person to another, and be copied and distributed. If someone likes it, that person can give a copy to a friend, but that friend will only see the summary until contacting the source to pay the fee. In order to avoid circumvention of this scheme, this active property would intercept another operation—the operation of trying to remove this property from the document! Active properties can be made active enough to be tamper-proof.[8]

[8] An example of this is a property we call "Immutable." Immutable stands as a guarantee that the document will never change. For example, when a manager signs a purchase authorization, the Immutable property ensures that the contents of the purchase order cannot be changed over his or her signature. The Immutable property fails to make this guarantee if it can be removed; so it uses a verifier for the `deleteProperty` operation that ensures that not only the content but also the properties (including the property of being Immutable) are immutable.

Active properties, then, not only simplify document use and management by end-users, but also assist developers by supporting a new model for the encapsulation and delivery of document services.

6.3 User-Specific Active Properties

Placeless Documents handles active and static properties seamlessly. This means that active properties have names and values, just like static properties; their values can be used, for example, to parameterize their behavior. In addition, just like static properties, active properties are user-specific.

Consider the document translation active property described earlier in the section on content transformation. The purpose of the translation property was to effect a transformation of document content for use by the consumer of the document. So, the translation that is appropriate at any given moment depends on *who* the consumer is. Different consumers need different transformations; some people need the content translated into French, into Spanish, etc. Just as with static properties, this is achieved by using document references to capture user-specific document properties. The properties recorded by document references can be active as well as static, and they will be invoked on relevant operations on the document reference. This allows each user to attach active properties that customize the document system behavior to his or her own needs, while still maintaining coordination through the base document content.

We described earlier the way that user-specific static properties allow users to organize and categorize documents according to their own specific needs, irrespective of how those documents are organized by others. User-specific active properties give users the same flexible control over document services and active behavior, moving computation closer to document consumers.

7. EXPERIENCES AND OPEN ISSUES

At the time of writing, our Placeless Documents infrastructure has been operational for over a year, and in daily use for around nine months. Our implementation is built entirely in Java 2, and runs on a variety of platforms and environments, including PCs under Windows NT and Linux, and SPARCs under Solaris. It supports a variety of protocols for document access, including HTTP, FTP, NFS, and IMAP. It supports a range of applications developed by ourselves and our colleagues, including workgroup document corpus management, email, workflow, and mobile document applications. Interactive response is fast enough to support novel direct-manipulation interfaces, and we regularly operate with personal document spaces numbering tens of thousands of documents and hundreds of thousands of properties.

In designing, developing, and implementing the Placeless Document system, a number of areas have been left open. These may be issues for which we simply do not yet understand the problems well enough to make

an effective design decision, or areas that we explicitly wish to explore using Placeless Documents as an infrastructure. Our early experiences with application development have highlighted others.

One issue that we have already touched on is the question of the structure of applications built on top of Placeless. Active properties, in particular, introduce the opportunity for new models of application development by supporting the migration of functionality from an application to the document itself. However, even simple static properties can introduce new models of application structure. In another paper [Dourish et al. 1999b], we discuss the experience of developing applications using static properties. One common observation is that the property model creates a duality in the infrastructure; it can act both as a document system (managing access to content, grouping documents for inspection, etc.) and a persistent object system (storing and searching metadata, linking representational structures, etc.). In one application, a tailorable collaborative document repository for a specific work group we have been studying, we explicitly exploit this duality not only to manage a large document collection but also to allow users to create and share customized views of the *structure* of that document space. The system can use our infrastructure to store both the documents and the encoded category structure descriptions through which the workgroup collaborates. The general question of the nature of "application" in the presence of an active infrastructure is one which we are exploring through the development of a variety of applications that exploit the features of Placeless Documents; for instance, Edwards and LaMarca [1999] discuss a set of applications in which application functionality has been devolved to a set of active properties.

The question of application structure also draws attention to the management of activity between client and kernel. In Placeless Documents, active properties run in the kernel, while delegates allow component functionality to migrate to clients. One common use of active properties is to allow clients to be informed of kernel-level activity; a recently added notification mechanism provides streamlined access to activity information. In developing applications on the Placeless Documents infrastructure, we are exploring the balance between kernel and client activity.

Another set of open issues concerns the level at which the features of the Placeless Documents system can be exploited by end-users. Again, the dual nature of Placeless as both document system and object system is relevant here; as well as offering system developers the ability to create new applications based on a uniform document property paradigm, Placeless also offers users the ability to exploit document properties directly to manage their own document collections. The Vista browser [Dourish et al. 1999a] allows users to manipulate and organize their documents directly in terms of properties, with multiple workspaces for supporting different tasks and direct manipulation of dynamic queries and fluid collections. Other applications focus on ways to incorporate information that reflects specific document practices in order to control the flexibility that a property-based approach offers and constrain it in ways that make it more

effective for particular tasks, particularly where those tasks involve the coordination of multiple individuals [Dourish et al. 1999c]. More generally, we are interested in how the multiple parallel organizational schemes allowed by document properties can support the fluid nature of everyday document categorization tasks [Harris and Henderson 1999]. The development of open and evolvable information structures must be balanced against the structure within the system itself that is the basis of high performance; the balance between these concerns is an important issue in our current activities.

Third, a number of issues at the infrastructure level also remain to be resolved. Document properties lie at the intersection of user and system concerns. Properties can be employed to specify user requirements and needs, and these needs can be exploited by the system infrastructure to provide appropriate levels of service. So, for example, document replication and caching can be informed by the properties used to express users' interest and expectations of documents; and active properties can be a means for the dynamic management of infrastructure services in response to document activity. As discussed earlier, our architecture has been designed to accommodate a wide range of replication schemes and styles. We have also begun to explore the use of active properties to support document content caching [de Lara et al. 1998] to improve responsiveness. A number of features in the Placeless Documents conceptual design, including active properties that transform content and per-user document properties, introduce new challenges for content caching. Our current implementation is designed to be highly configurable so that we can explore effective approaches in the context of our different applications.

Finally, based on early development experiences, we are considering ways to refine the APIs for application development. A number of possible directions have emerged from our early explorations. One issue of particular interest is the use of semistructured schemas that programmers can impose over the property store in order to express invariants and constraints for document objects that represent application state. This allows programmers to incorporate higher-level information and automate some aspects of data management in their applications, without giving up the flexibility to create arbitrary new categorizations of documents and to customize their behavior using active properties. In addition, this information may be used at the database level to adaptively adjust the low-level data organization for particular application workloads.

8. CONCLUSIONS

Documents in most systems are organized for the convenience of the systems rather than their users. Such systems use strict hierarchies to organize documents, and require users to adapt their document practices to the structures that they impose. They make it hard for people to file, locate, and share documents. Moreover, they separate documents from the activities that users perform over them, encapsulated in different applications.

The Placeless Documents system introduces the concept of personalized document properties as a uniform mechanism for organizing, filing, grouping, retrieving, and manipulating documents and document collections. Using properties as the fundamental mechanism for user interaction with documents provides a number of benefits in comparison to traditional hierarchical filing structures.

First, it allows documents to be managed *for the document consumer*. Since different users can have different properties on the same document, the control over the document and its use passes from the document's author to the many different consumers who may make use of it. Second, it creates a single, uniform interaction model for a wide range of information about the document. Information which would otherwise be locked inside different applications (such as the titles of a presentation slides, the subject lines of mail messages, and the classes defined by Java source files) can be extracted and brought together in a single space and operated on by common actions. Third, it reflects the "multivalent" nature of documents. A single document may be relevant to users in a number of different ways, reflecting the different roles it plays and the different tasks users might want to perform. Since any number of properties may be attached to a document, users can organize their documents into multiple cross-cutting organizational structures.

The Placeless Document system is based on *uniform interaction*, *user-specific properties*, and *active properties*. Uniform interaction stresses a single global property space, integrating information from many sources. User-specific properties allows users to organize and control shared documents in ways that are relevant to them individually. Active properties carry code to be performed when specific operations are carried out on the documents to which they are attached. Active properties can make documents responsive to the activities over them; they can be used to configure and control system services; they can be used to deliver document services seamlessly within the document management system; and they do all this within the same uniform document property framework.

The incorporation of these features into a document management system poses a number of conceptual and practical challenges. In this article, we have described how we addressed these challenges in Placeless Documents, a distributed, user-centric document management system that we developed as an infrastructure for property-based applications. Placeless Documents is designed to be a scalable, reliable, distributed platform for document management.

We are currently exploring the consumer-focused active property paradigm in a number of ways. First, we are developing a range of applications that run on the Placeless Documents infrastructure, to explore how active properties can be used to structure applications, associating functionality directly with documents and giving users the means to directly configure and control it. Second, we are investigating how infrastructure services such as document replication and content caching can be controlled by

active properties. Third, we are exploring how active properties can be used to give end-users direct control over composable document services.

Our early experiences with Placeless Documents, as users and developers, suggest that it is a powerful infrastructure for the development of novel document applications.

ACKNOWLEDGMENTS

We would like to thank a number of colleagues and collaborators for contributions to the research described here, including Danny Bobrow, Dan Greene, Mike Spreitzer, Mark Stefik, Dan Swinehart, and Marvin Theimer. Our interns, Dirk Balfanz, Jon Howell, Eyal de Lara, and Minwen Ji, contributed to the development of our prototype implementation, and early users including Ian Smith, Tom Rodden, Michelle Baldonado, and Jacek Gwizdka provided valuable feedback.

REFERENCES

AHLBERG, C., WILLIAMSON, C., AND SHNEIDERMAN, B. 1992. Dynamic queries for information exploration: An implementation and evaluation. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI '92, Monterey, CA, May 3–7), P. Bauersfeld, J. Bennett, and G. Lynch, Eds. ACM Press, New York, NY, 619–626.

BARREAU, D. AND NARDI, B. A. 1995. Finding and reminding: File organization from the desktop. *SIGCHI Bull. 27*, 3 (July 1995), 39–43.

BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C., AND EGGERS, S. 1995. Extensibility safety and performance in the SPIN operating system. *ACM SIGOPS Oper. Syst. Rev. 29*, 5 (Dec.), 267–283.

BIER, E. A., STONE, M. C., PIER, K., BUXTON, W., AND DEROSE, T. D. 1993. Toolglass and magic lenses: The see-through interface. In *Proceedings of the ACM Conference on Computer Graphics* (SIGGRAPH '93, Anaheim, CA, Aug. 1–6, 1993), M. C. Whitton, Ed. ACM Press, New York, NY, 73–80.

BOROWSKY, E., GOLDING, R., MERCHANT, A., SCHREIER, L., SHRIVER, E., SPASOJEVIC, M., AND WIKLES, J. 1997. Using attribute-managed storage to achieve quality of service. In *Proceedings of the Fifth International Workshop on Quality of Service* (New York, NY),

BOWKER, G. AND STAR, S. 1994. Knowledge and infrastructure in international information management: Problems of classification and coding. In *Information Acumen: The Understanding and Use of Knowledge in Modern Business*, Bud, Ed. Routledge & Kegan Paul Ltd., London, U.K..

CHAMBERS, C. 1992. Object-oriented multimethods in cecil. In *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP'92, Utrecht, Belgium), Springer-Verlag, Berlin, Germany, 33–56.

CODY, W. F., HAAS, L. M., NIBLACK, W., ARYA, M., CAREY, M. J., FAGIN, R., FLICKNER, M., LEE, D., PETKOVIC, D., SCHWARZ, P. M., THOMAS, J., ROTH, M. T., WILLIAMS, J. H., AND WIMMERS, E. L. 1995. Querying multimedia data from multiple repositories by content: the Garlic project. In *Proceedings of the third IFIP WG2.6 working conference on Visual database systems 3 (VDB-3)*, S. Spaccapietra and R. Jain, Eds. Chapman and Hall, Ltd., London, UK, 17–35.

DOURISH, P., EDWARDS, K., LAMARCA, A., AND SALISBURY, M. 1999a. Presto: An experimental architecture for fluid interactive document spaces. *ACM Trans. Comput. Hum. Interact. 6*, 2.

DOURISH, P., EDWARDS, K., LAMARCA, A., AND SALISBURY, M. 1999b. Uniform document interaction with document properties. In *Proceedings of the ACM Symposium on User Interface Software Technology* (UIST '99, Asheville, NC, Nov.), ACM, New York, NY.

DOURISH, P., LAMPING, J., AND RODDEN, T. 1999c. Building bridges: Customisation and mutual intelligence in shared category management. In *Proceedings of the ACM Conference on Supporting Group Work* (GROUP '99, Phoenix, AZ), ACM, New York, NY.

EDWARDS, K. AND LAMARCA, A. 1999. Balancing generality and specificity in document management systems. In *Proceedings of the 7th IFIP Conference on Human-Computer Interaction* (INTERACT '99, Edinburgh, Scotland), IFIP, Laxenburg, Austria.

FREEMAN, E. AND FERTIG, S. 1995. Lifestreams: Organizing your electronic life. In *Proceedings of the AAAI Fall Symposium on AI Applications in Knowledge Naviation and Retrieval* (Cambridge, MA, Nov.), AAAI Press, Menlo Park, CA.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

GIAMPAOLO, D. 1998. *Practical File System Design with the Be File System*. Morgan Kaufmann, San Mateo, CA.

GIFFORD, D., JOUVELOT, P., SHELDON, M., AND O'TOOLE, J. 1991. Semantic file systems. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles* (Pacific Grove, CA), ACM Press, New York, NY.

GREENE, S. L., DEVLIN, S. J., CANNATA, P. E., AND GOMEZ, L. M. 1990. No IFs, ANDs, or ORs: a study of databases querying. *Int. J. Man-Mach. Stud. 32*, 3 (Mar. 1990), 303–326.

GUY, R. 1990. Implementation of the Ficus replicated file system. In *Proceedings on Summer USENIX Conference* (June 1990),

HARRIS, J. AND HENDERSON, A. 1999. A better mythology for system design. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI '99, Pittsburgh, PA, May), ACM Press, New York, NY.

KIM, W. 1990. Object-oriented databases: Definition and research direction. *IEEE Trans. Knowl. Data Eng. 2*, 3 (Sept.), 327–341.

KISTLER, J. J. AND SATYANARAYANAN, M. 1992. Disconnected operation in the Coda File System. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 3–25.

LAMARCA, A., EDWARDS, K., DOURISH, P., LAMPING, J., SMITH, I., AND THORNTON, J. 1999. Taking the work out of workflow: Mechanisms for document-centric collaboration. In *Proceedings of the 6th European Conference on Computer-Supported Cooperative Work* (ECSCW '99, Copenhagen, Denmark, Sept. 12–16), Kluwer Academic, Dordrecht, Netherlands.

DE LARA, E., PETERSEN, K., TERRY, D., LAMARCA, A., THORTON, J., SALISBURY, M., DOURISH, P., EDWARDS, K., AND LAMPING, J. 1998. Caching documents with active properties. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems* (HOTOS-VII),

MOGUL, J. 1984. Representing information about files. In *Proceedings of the Fourth International Conference on Distributed Computing Systems* (San Francisco, CA), IEEE Press, Piscataway, NJ, 432–439.

PATON, N. W. AND DIAZ, O. 1999. Active database systems. *ACM Comput. Surv. 31*, 1, 63–103.

PHELPS, T. A. AND WILENSKY, R. 1996. Toward active, extensible, networked documents: Multivalent architecture and applications. In *Proceedings of the 1st ACM International Conference on Digital Libraries* (DL '96, Bethesda, MD, Mar. 20–23), E. A. Fox and G. Marchionini, Eds. ACM Press, New York, NY, 100–108.

RICHTER, J. AND CABRERA, L. F. 1998. A file system for the 21st Century: Previewing the Windows NT 5.0 file system. *Micr. Syst. J.* (Nov.).

SUN MICROSYSTEMS. 1989. Network file system protocol specification (RFC 1049). DDN Network Information Center, SRI International, Menlo Park, CA.

TRIGG, R., BLOMBERG, J., AND SUCHMAN, L. 1999. Moving document collections online: The evoluation of a shared repository. In *Proceedings of the 6th European Conference on Computer-Supported Cooperative Work* (ECSCW '99, Copenhagen, Denmark, Sept. 12–16), Kluwer Academic, Dordrecht, Netherlands, 331–350.

UNGAR, D. AND SMITH, R. B. 1987. Self: The power of simplicity. In *Proceedings of the OOPSLA 1987 Conference on Object-Oriented Programming Languages, Systems and Applications* (OOPSLA'87), ACM, New York, NY, 227–242.

Received: February 1999; accepted: December 1999