A Programming Model for Active Documents

Paul Dourish, W. Keith Edwards, Jon Howell, Anthony LaMarca, John Lamping, Karin Petersen, Michael Salisbury, Doug Terry and Jim Thornton

Computer Science Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto
CA 94304 USA
paul@dourish.com

ABSTRACT

Traditionally, designers organize software system as active end-points (e.g. applications) linked by passive infrastructures (e.g. networks). Increasingly, however, networks and infrastructures are becoming active components that contribute directly to application behavior. Amongst the various problems that this presents is the question of how such active infrastructures should be programmed.

We have been developing an active document management system called Placeless Documents. Its programming model is organized in terms of properties that actively contribute to the functionality and behavior of the documents to which they are attached. This paper discusses active properties and their use as a programming model for active infrastructures. We have found that active properties enable the creation of persistent, autonomous active entities in document systems, independent of specific repositories and applications, but present challenges for managing problems of composition.

Keywords: Active properties, document management, component software, customization.

INTRODUCTION

As computer systems become more powerful and network bandwidth and capacity increases, new models are emerging for the development of infrastructure technologies. One of these is what we call "active infrastructures."

Traditional approaches have typically concentrated computational power in fixed locations. So, for example, the mainframe approach concentrates computing power in one large, centralized system. Client/server computing distributes it between two points, but regards the channel between those points – the network or infrastructure that connects them – as a static channel. Active infrastructure approaches explore the opportunities to devolve some computation into the infrastructure itself. In this model, application semantics can migrate into infrastructure, which itself becomes an active entity that can specialize itself to the needs of different applications. Active infrastructures have been explored in a range of domains. For example, the Bayou distributed database system allows database updates to carry with them procedures that can resolve conflicts that are encountered as they move through the network [24]; or again, active networking allows code to be "injected" into the network so that routers and other network components can be specialized to the needs of different applications [27]. Active infrastructure approaches provide a number of advantages. They provide applications with specialized infrastructure arrangements providing cleaner implementation models; they make more efficient use of infrastructures by incorporating specialized facilities rather than working only in terms of generic features; and they allow infrastructures to adapt to variations in application demands.

Since active infrastructures are a new approach, however, they are unfamiliar to programmers, and are not directly supported by conventional programming tools (be those "conceptual" tools or standard software tools). A set of questions arise, then, about the programming model through which active infrastructures can be presented and controlled.

In the Placeless Documents project, we have been exploring an active infrastructure approach to the provision of document and document management services. Our approach is based on a distributed infrastructure in which activity can be directly associated with documents, rather than being locked inside applications that are invoked to process those documents. By pushing activity into the infrastructure, we can make it independent of particular repositories and applications, so that users can organize activity around their tasks rather than around the details of applications. The Placeless Documents design explores compositional approaches to document service functionality, and new relationships between applications, infrastructures, and services.

Research Questions

Our system is designed as an infrastructure for interactive document applications. Active infrastructure approaches are normally visible only to systems programmers; their use at the application level is relatively novel. This opens up two sets of questions that this paper will address.

The first is how active infrastructures and the extensibility techniques they introduce can be incorporated into an interactive system model. What sort of conceptual model can be

UIST '00. San Diego, CA USA

offered to end users to understand how activity is incorporated into the infrastructure, and how can this be incorporated into a component model that allows different activities to be present at the same time?

The second (related) issue is that of the programming model that the system will present. How can we combine interactive system programming with active infrastructures? What consequences does an active infrastructure approach hold for interactive system design and how can these design concerns be manifest to programmers? How can the active infrastructure approach be incorporated into current design practice?

In this paper, we reflect on the experiences of designing the Placeless Documents infrastructure and developing applications on top of it. We explore the programming model that we developed and some of its consequences, and show how it was exploited in applications that we and others developed. On the basis of these experiences, we draw out some lessons and discuss our current approach to the provision of active infrastructure in follow-on work.

PROPERTY-BASED DOCUMENT INFRASTRUCTURE

The Placeless Documents system has been in development since late 1997. By this stage, we have gained experience with a variety of prototype implementations as well as a range of applications of different styles, scopes and models.

The name Placeless Documents reflects the core of our underlying motivations. Most information management systems employ hierarchies as the dominant paradigm for information management – files and directories, email messages and folders, etc. Hierarchies are used to perform multiple functions. They are used to present information; they are used to retrieve information; they are used to store information; and they are used to control information. So, for example, when I store a document in the filesystem, I put it at some particular place in the filesystem hierarchy that both reflects some features of the document (e.g. when I put it in `T:\home\papers\drafts\uist\placeless.doc`) and where I think I will remember to look for it again; and, by putting it in certain places (e.g. the Microsoft Windows "briefcase"), I control something of how that document behaves.

In our model, we want to separate the expression of document features and document control from the system of "places" that the hierarchy describes (hence, "Place-less"). Our alternative model is based on document *properties*. Properties are document metadata tags that users and applications can associate with documents. They are implemented as arbitrary pairs of string names and Java object values; their values can be set, tested, retrieved and searched. A document can have many different properties associated with it. We use properties to encode information that is relevant to the users of the documents (e.g., that a document is a paper, that it is a draft, that it is being prepared for UIST, etc.) as well as to associate application information with documents (e.g. the history of application actions over that document). Users add properties to documents either directly through drag-and-drop interfaces such as those explored in a previous UIST paper [6], or indirectly through property-based applications [10, 14].

Active Properties

We introduced the document property model in a previous UIST conference, where we outlined the development of our initial prototype, called Presto[1] [6]. Presto used document properties to provide a persistent associative document store for end users and applications, but in Presto the store was entirely static. In the full Placeless Documents system, the property store is used as an active infrastructure through the addition of active properties.

Like attributes in Presto, Placeless' active properties include both a name component and a value component. However, active properties include a third component – runnable code. This code is designed to be run inside the infrastructure in response to various actions upon the document. By attaching active properties to a document, users can make the document responsive to the situations in which it is used. Since a document can have any number of properties attached, active properties provide users with compositional control over document behaviour.

What sort of behaviours can users achieve through active properties? Active properties can ensure that document are automatically backed up, or are maintained consistently in multiple places (e.g. on a laptop and a server). Active properties can take functionality normally associated with specific applications such as workflow, format or content conversion or specialised presentation, and associate them directly with the document so that they travel with the document wherever it goes, as it is emailed around, transferred between systems and so on. Active properties can automatically detect document content changes and implement features such as notification, summarisation, or version control. Further, these can all be controlled compositionally, available over all document under user control.

We have built the Placeless Documents infrastructure and used it extensively, exploring its opportunities by building property-based applications and functional elements including those described above. Elsewhere, we describe the technical concerns in extending a document management system to incorporate active properties, including the distribution and efficiency issues that are involved [7]. In this paper, however, we are concerned with the programming model that active properties offer, and with the ways in which infrastructure activity can be encapsulated and provided to application developers.

THE ACTIVE PROPERTY PROGRAMMING MODEL

The Placeless Documents infrastructure is written in Java. We provide a number of interfaces to interoperate with existing application infrastructures. For instance, we offer access to the Placeless Documents repository through HTTP, so that existing web clients can operate with Placeless Documents; similarly, we provide access through other standard Internet protocols such as IMAP and FTP, and offer an NFS interface so that filesystem-based clients can gain access to Placeless without any adaptation. New applications, includ-

1. Presto used the term "attribute" rather than "property," but otherwise its model is a subset of the property model in Placeless Documents.

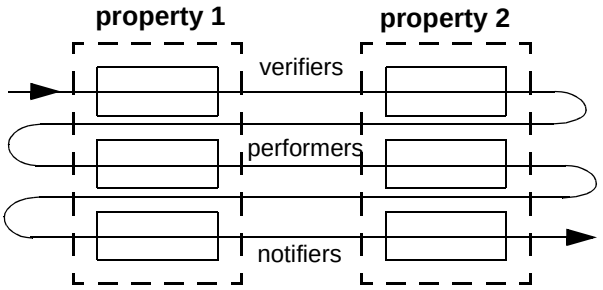

Figure 1: Phased execution of active property code.

ing new services to be written as active properties, are typically written in Java, using custom APIs.

The infrastructure provides two sorts of active properties, *inline* active properties and *delegates*.

Inline Active Properties

The standard form of active property is an *inline* active property. Inline active properties change document behaviour by intercepting and inserting themselves into the execution path of document operations such as `deleteDocument`, `addProperty`, `readContent`, and so on.

Placeless provides twelve of these core operations. Each active property can be associated with any number of these operations. An active property that transforms content might insert itself into the execution path of both the `readContent` and `writeContent` operations, so that it can transform content symmetrically; one that logs all document activity might insert itself into the execution path of *all* the document operations.

Combining Active Properties

A document can have many properties, and since active properties act just like normal properties, it follows that a single document can have multiple active properties. Since any active property can intercept any set of document operations, it follows that a document might have more than one active property interested in a specific operation. For example, the setProperty operation might be of interest to two different active properties: an access control property that wants to restrict write access to the document, and a logging active property that wants to maintain a history of document activities. Our infrastructure, then, must provide some mechanism for controlling how these active properties combine.

There are two mechanisms that control property combination: *property ordering* and *phased execution.*

In phased execution, the dispatch cycle for a single operation on a document is divided into three phases, called the *verify*, *perform* and *notify* phases. Conceptually, the verify phase determines that the operation is allowed; the perform phase carries out the operation; and the notify phase carries out any post-execution cleanup and notifications. When an active property inserts itself into the execution path for an operation, it specifies which phase it should be associated with. Although an active property may have many code methods, each method is associated with just one phase.

The overall model is shown in figure 1. Consider a specific document operation. A user or application has invoked the `addProperty` operation on a document, to add a new property to it. This operation dispatches into the Placeless infrastructure, where the active property dispatcher takes over. First, the dispatcher scans the active properties attached to this document to determine which have associated themselves with the `addProperty` operation in the verify phase. This results in a set of methods, each of which is defined to take as arguments the set of arguments for the `addProperty` operation, and return a boolean value. Each of these methods is called in turn. If any of the functions returns false, then the execution sequence is terminated and an exception is thrown to the calling application, informing it that an active property has declined the operation. Otherwise, the dispatcher advances to the perform phase.

Again, each property is examined in turn, this time for performer methods. The `addProperty` operation does not support performers, but for those operations that do, the performer methods may add functionality that alters the effect of the original operation and its return value if there is one. The interfaces are arranged so that each perform-phase property has access to the result computed "so far" by the other performers on the same document. For example, an output stream for writing the document content will be built so that each interposing active property may alter the content as it flows down the stream.

Finally, the dispatcher advances to the notify phase. Again, the active properties that have registered themselves for the notify phase of the `setProperty` operation are called in turn. Notify methods have no return value; they are entirely independent of each other. Once they have all been executed, the return value computed during the perform phase is retrieved and returned as the outcome of the operation.

Phased execution allows us to control some of the effects of combining properties, by allowing programmers to associate

Table 1: Operations and their Active Properties

	Verify	Perform	Notify
AddProperty	•	-	•
DeleteProperty	•	-	•
AddMember	•	-	•
GetMembers	•	•	•
RemoveMember	•	-	•
GetPropertyValue	•	-	•
SetPropertyValue	•	-	•
SetQuery	•	•	•
DeleteDocument	•	•	-
ReadContent	•	•	•
WriteContent	•	•	•
CloseOutputStream	-	-	•
GetDelegateFor	•	-	•

them with specific phases of execution. However, there are other elements of property interaction, particularly in the case of side effects. We will discuss these, along with the property ordering mechanism, in more detail after laying out the full programming model.

Writing Active Properties

Active property writers create a Java class that implements the `ActiveProperty` interface. This interface requires some standard methods for initializing the active property object itself. In addition, this class will implement a number of other interfaces. These interfaces describe the operations that the active property will intercept. For example, the class for an active property that wants to intercept the verify phase of the `setPropertyValue` operation and the notifier phase of the `writeContent` operation will implement the interfaces `SetPropertyValueVerifier` and `WriteContentNotifier`. Table 1 shows the set of available interfaces that an active property writer can use. Each interface defines the methods that will be called by the dispatch engine at the relevant point in the dispatch cycle.

By providing inline active properties that intercept and redirect document operations, programmers give end users control over the interactive behavior of their documents. Since active properties can be added to documents at any point in their lifecycle, this control can be exerted at any moment and continually revised. From the programmer's perspective, code can be incorporated automatically and transparently with no prior knowledge on the part of the document or application developers.

Delegates

The second sort of active property in the Placeless system is the delegate. While inline active properties insert themselves into the execution path of existing operations, delegates extend the API of the documents to which they are attached, providing new functionality and new call paths.

Conceptually, a delegate is an object which stands for the document with respect to some operation (in fact, with respect to a Java interface). If an application wants to make use of a potential document interface extension for a specific document, then it makes a call on the document object, requesting a delegate that implements the interface. The document returns a delegate that implements the requested interface, and that stands for the document for the purposes of that interface. For example, there is no `Backup` operation defined in the standard document operations. However, a backup application might call a method on a document to request a delegate that implements the `BackupableDocument` interface. If the document is capable of providing one, it returns an object that implements that interface; calling the method `getLastBackupTime()` on the delegate would report when that document was last backed up, and calling `backup()` on it would place a copy of the document on a stable backup medium.

There are three important features to note concerning the programming model offered by delegates.

The first is that, while the execution of inline active properties is carried out in the Placeless infrastructure, the execution of delegates is carried out in the application. That is, the delegate object is returned to the application and becomes part of the application's address space, with the application in control of when the methods on that object are invoked. In contrast, inline active properties are executed as a part of the normal execution of the system, and so must run in the Placeless Documents core since they must operate across all applications.

The second feature is that delegates require coordination between the document and the application. Delegates are only provided when an application knows to ask for them, and knows what interface is required. Delegates do not become active for unspecialized applications.

The third important feature of delegates is that they are a type-safe extension mechanism. Applications request a delegate specifying a specific Java interface. The delegate that is returned is an instance of a class that implements that interface, and is cast to the interface type. Calls upon that interface can then be made directly, and can be type-checked by the Java compiler. In contrast to other extension mechanisms that might rely on the use of features such as reflection to map strings into method names, this approach can exploit the language's type system.

Implementing Delegates using Inline Active Properties

Although we have contrasted delegates with inline properties here, inline active properties are in fact the mechanism by which delegates are associated with documents.

One of the basic document operations that an inline active property can specialize is called `getDelegateFor()`. This method takes a Java interface as its argument and returns an object that implements that interface as a result. So, a programmer wishing to create the `Backup` delegate would create two classes. The first class is an implementation of the `Backup` interface, and performs the specialized backup operations. The second class is an active property that specializes the `getDelegateFor()` method and returns an instance of the implementation class when it determines that the relevant interface is being requested.

One of our design issues is how a single model supports both programming and end-user interaction. Using inline active properties to implement delegates allows users to extend document behaviors through the same mechanism – attaching properties to documents – that they use to perform all other customizations of the system.

Delegates and Object-Oriented Delegation

Delegation is a technique that has already been used to great effect in object-oriented programming systems. In fact, prototype-based object systems often employ delegation as a means to achieve the same effects that can be achieved using inheritance in class-based systems [21]. Our use of delegation is similar, but we use the term in ways that do not match the conventional structure of object-oriented delegation. In an OO delegation system, a message sent to one object may be "delegated" or redirected to a second object, which will execute the method associated with that message on behalf of the first object (or further delegate it). The result of this method invocation will be returned to the object that origi-

nally sent the message. This can be seen to be structurally equivalent to method inheritance; imagine that, by delegating the message, the first object seems to inherit from the second object the ability to respond to the message.

Our use of the term "delegate" differs from this model in three ways. First, our delegates are explicitly visible to the application; we do not automatically delegate on message sends. Second, we delegate "downwards" rather than "upwards." In our model, a generic object (a document) provides a more specialized object (the delegate) which will respond to a set of messages on its behalf. Third, the delegate does not appear to "subclass" the document because it does not, itself, implement the `Document` interface; instead, it implements only the interface specified by the application.

Although the differences from OOP conventions are confusing, the separation of the delegate functionality from the document functionality provides some benefits in the distributed environment of Placeless programs. In particular, it means that the delegate can be shipped to other processes or other nodes without creating the confusion about the identity of the document that might result from two "document" objects on different machines.

ACTIVE PROPERTIES AND INTERACTIVE BEHAVIOURS

One feature of the Placeless Documents system and its approach to active documents that distinguishes it from earlier explorations of active infrastructures is that Placeless is primarily an infrastructure for interactive applications. The interaction aspects of our model have two sets of consequences.

First, the control that active properties themselves offer over document behavior is interactive control. Active properties can be added, removed and controlled by end users. Indeed, in some of our browsers, active properties are entities that users simply drag-and-drop in order to change the behavior of documents. This means that the procedures by which active properties can be associated with documents must be both simple and responsive, that active properties must be consistent in their interactions, and that active properties must support arbitrary compositions. Stronger class-based approaches, for example, would not satisfy these requirements.

Second, active properties may affect the interactive behavior of documents. This affects the *style* of development; since documents are interactive entities, active properties must support interactive response times. In addition, active properties themselves sometimes manage other interactive objects, such as when they "decorate" document interfaces with buttons and widgets corresponding to currently-available actions. The interactive requirements entail a lightweight approach to active property programming, which in turns leads to a style of development involving multiple interacting active properties (which we will discuss in more detail in the discussion of Programming Issues).

EXAMPLES OF ACTIVE PROPERTIES

Active properties provide a flexible infrastructure for associating behavior with documents. To make some of this discussion more concrete, we will briefly describe some of the applications we have developed using active properties, and then go on to explore some of the issues that arise in using active properties as a programming model for active documents.

Workflow

One series of developments concerned the provision of workflow and document management services via active properties [5, 14]. Providing workflow in this way migrates it into the infrastructure, and makes it independent of either specific applications or specific repositories.

Our approach combines both inline active properties and delegates. A delegate encapsulates a workflow-specific API that allows an application to explore the process instances with which a document is associated, query their state, progress a document from one state to another, associate notifications and so forth. Since it is provided through a delegate, this functionality can be highly specialized but is available only to special-purpose applications that understand how to call on it. By using an inline active property, we can connect workflow functionality with existing applications. An active property is associated with the read and write operations for the document; it notices when those operations are performed and passes notifications to the delegate, which can analyze the changes to the content and associate them with the workflow process. So, for example, if a document represents a form with check boxes, the active property can notice when the check boxes have been selected, and the delegate can cause the document to be moved to the next stage of the process, accordingly.

Our implementation uses an internal workflow engine to represent and control process instances. However, many commercial workflow systems are organized according to the reference model developed by the Workflow Management Coalition and provide network-accessible mechanisms through protocols such as SWAP [22]. A simpler way to write our delegate would be to make it a client of such a service. In this way, we could use active properties to coordinate document activity with an external workflow service, but provide this "activation" independent of any particular application that end users might want to use.

Delivering Services

The idea of using active properties to coordinate document action with external mechanisms such as workflow leads to a variety of ways in which active properties can be used to deliver document services.

Obvious examples include format conversion (e.g. from Microsoft Word to PDF) and interpolation (e.g inline recoding of images to reduce bandwidth requirements [13]). We have also used these mechanisms to incorporate higher-level document services such as content filtering, summarization and language translation.

Delivering services such as these through active properties offers two advantages over conventional approaches, one technical and one interactive. The technical advantage is that the services can be offered independently of application or repository. The service is delivered at a point between the repository and the application, and so applies to any combi-

nation. The interactive advantage is that end users can compositionally control the deployment of services on a document by document basis. Since the functionality appears to be associated directly with documents, it makes sense to allow users to control it by acting directly on those documents, and so, notionally, control the behavior of the documents rather than that of an abstract service.

Versioning

Finally, we have also used active properties to augment the services traditionally associated with repositories. For example, using active properties, we can add versioning to a repository that does not otherwise support it.

There are a number of implications of adding versioning via active properties rather than building it directly into the infrastructure. For instance, since the Placeless infrastructure is unaware of versioning, it provides no direct support for the way in which versioning makes document identity more complex. The infrastructure, for example, will not be able to recognize that two different versions are actually the "same" document. A second consequence is that, since versioning is added at the middleware level rather than the repository level, we can, in fact, take advantage of underlying versioning facilities when they are provided. This flexibility comes at a cost; since different repositories often have different version semantics, we need to be able to interpret and interpolate between them.

The versioning property maintains a chain of documents that are earlier versions of the current content. Each time a user opens the document for writing, a copy of the original content is made and linked to the document as an "earlier" version. This is done by attaching to the document an active property that intercepts the `getOutputStream()` operation, and hence notices all attempts to write new versions of the content.[2] Previous operations can be retrieved either by looking directly at the properties that link a document to previous versions, or through a delegate property which adds an API for reviewing and retrieving earlier versions.

PROGRAMMING ISSUES

The Placeless Documents infrastructure first became operational during the summer of 1998; since that time it has been in daily use and we have refined and revised our core designs significantly. We have also gained considerable experience with active property-based applications, some of which we developed ourselves, and some of which have been developed by colleagues elsewhere at PARC. A variety of programming issues have arisen from our experiences developing applications with active properties.

Programming Using Static Properties

Before discussing active properties, we should first explore how simple static properties impact programming style. The combination of freely extensible static properties and fast query mechanisms allow programmers to exploit new models for structuring their applications.

Properties provide convenient associative storage. Information can be stored alongside the documents to which it applies, and retrieved by queries. At the same time, because property objects store not just primitive types but arbitrary serialized Java objects, a document's properties can point to other documents, and so on, allowing programmers to create complex data structures as sets of related documents. Having a document store that can be used as a persistent object store, programmers intuitively adopt a style in which data structures are distributed across documents, stored persistently, and reconstituted through queries over the document space.[3] Most importantly, since properties are compositional, a single document may participate in many different applications or data structures. The compositional use of static properties is mirrored in the use of active properties.

Creating Responsive Documents

One way of interpreting the effect of active properties is to consider that operations that would otherwise be fixed in their consequences can now be made open and flexible. So, for example, whereas reading a document's content from disk and displaying it in a window is normally a fixed operation with a fixed implementation, active properties give users and documents individualized control over these operations. The result is that documents can be made responsive to the contexts in which they are used. A trivial example is that document content can be transformed according to the person who reads it or the time at which it is read; similarly, other document operations can be made responsive to the contexts in which they are carried out.

This facility, along with the associating storage facilities provided by static properties, makes Placeless Documents an excellent platform for the development of interactive applications that exploit contextual factors such as location or participants [1, 19]. Static properties encourage a context-based approach in which documents and objects are annotated with information that reflects how they have been used, where, when, by whom, etc.; combinations of these properties can serve as retrieval cues or can be used to extract relationships between documents or application objects.

Using static properties, however, means that these contextual features can be exploited only when specific applications are running. Using active properties, context dependence can be migrated into the infrastructure. The contextual behavior is associated directly with the documents themselves. This is valuable since exploiting context is a key element of the ubiquitous or pervasive computing programs, and so requires support at the infrastructure level rather than in application space. Active properties provide a novel means for making "passive" entities into active elements of a ubiquitous computing environment. Through their active properties, documents can be made responsive to different aspects of their use. In addition to being responsive to the person who acts on them, documents might also be made

2. Intercepting property operations also allows the property to keep track of changes to the set of properties associated with the document, but we focus on content operations here.

3. Recognizing this feature of a number of early applications, we provided specialized support for it by developing a package that allowed programmers to reflect documents as Java Beans and vice versa.

responsive to other aspects of the context in which they are used; e.g. rendering themselves differently and with different interface options depending on the viewing device, on the time of day, or at different points in an organizational process [10].

Combining Inline and Delegate Properties

In our initial proposals for active property applications, we favored inline active properties. We proposed the use of active properties for document format transcoding, for mobile document services, for configuring the behavior of external applications or services, and for specializing the service characteristics of the infrastructure to application needs. Our early application development experiences, however, showed the power of delegates for exploring new application opportunities. So, our applications emphasized the way in which users could extend and augment document behavior using delegate properties.

In fact, there is an important duality between the two forms of active properties. Inline active properties intercept document operations, while delegates provide new facilities. A common active property idiom is to actually use both sorts; use an inline property to observe that some operation has taken place, and then activate a delegate to run some new document behavior in response. The workflow service is an example of this idiom. This pairing relies on a natural separation of "new" code from "interposed" code. It *could*, of course, be written as a single active property, but the use of two different sorts of active property seems to more accurately reflect the programmer's expectations.

Exploded Applications

In conventional systems, functionality is restricted to applications. We have shown that active properties allow functionality to be directly associated with documents and moved into the infrastructure. So, the presence of active properties causes us to reassess how applications work and how they are structured. In Placeless, we can start to think of applications as consisting of a variety of active properties that may be spread throughout the document space. We call these "exploded" applications.

As an example, consider a system that supports document linking, such as a hypertext system or a document editor that supports the inclusion of image files by reference. In a conventional application, the relationship between the documents is only active when the application is running. If a user deletes or moves a linked file, the application will not know; the result is a dangling link. Active properties provide a mechanism to prevent this problem. In Placeless, an application that supports document linkage can attach an active property to any linked document that will intercept move or delete operations. When these operations occur, it can notify the user that this is a linked document (and so the user may not want to move it), and/or notify the application that the document has been moved (and so it should update its pointers). The application has been "exploded" or spread throughout the infrastructure; it can now be active even when the central application is not running. So, active properties change our notions of what constitutes an application.

Ordering

One important set of issues arise around the ordering of property invocations. Properties provide compositional control over the behavior of a document, and ordering affects how their interactions are controlled.

Placeless provides two mechanisms to control ordering. The first is the three-phase model described earlier, where properties are invoked separately according to three roles (verifier, performer, notifier). This mechanism moves some of the more obvious potential property interactions (e.g. a property that wants to veto the attempts of another to perform an operation) into the structural domain of the infrastructure, rather than having the properties "fight it out amongst themselves." It also provides a more fine-grained model which, in turn, encourages property writers to work at a more fine-grained level.

However, it leaves many problems unaddressed. In particular, we observe conflicts between two active properties involved in the same phase of the same operation. We explored a variety of designs for this problem, and eventually fixed on a straight-forward numerical ordering for the invocation sequence of properties. This allows us to combine properties that have interdependent effects. For example, if we wanted to add to the same document one property that encrypted file contents before they were written to disk, and another that compressed them, we would want to ensure that they were always invoked in the right order.

A numerical ordering is clearly flawed in a number of ways; it requires the properties themselves to manage the potential negotiation to establish their relative order, rather than handling it automatically. However, we believe more complicated schemes to be overly complex; to establish a language of property side-effects, for instance, would make our API considerably more complex.[4]

Notifications

Another idiom that we observed in early applications was the use of notifications. "Notifier"-phase active properties had been included to support a variety of tasks such as audit trail logging and operation post-processing that could be conceptualized as a "notification" from one element of the system to another. However, more explicit notifications or callbacks between different applications turned out to be more common that we had anticipated. In particular, we found ourselves frequently writing notifier active properties – which run in the "kernel" or server – that simply looked up client processes and informed them of the event. What is more, for certain kinds of applications, such as applications that present a representation of the activity of other clients or over workspaces, we found these sorts of notifications being added to many documents. Placeless provided a mechanism for server-side notifications, but not for client-side notifications.

We augmented our basic APIs to provide support for client notifications. Clients can register their notifications with a

4. In fact, however, as we will discuss, there are some other benefits to having such a descriptive language.

server; should the client disappear before the notification is called, then the notification will be silently removed on the server side. Client-side notifications are associated with patterns of documents, properties or operations. So, an application can register a single notification that will apply to any number of documents, or to activities to groups of properties on those documents, and so on. In addition to reducing the number of notification instances[5] this facility also allowed notifications to be registered that cannot be attached to a specific document, such as a notification that a new document has been created.

Introspection

Our experiences with the compositional effects of active properties lead to a recognition of the importance of property introspection – the ability to examine and reason about the internal structure and behavior of active properties.

There are two reasons that we require some sort of introspection facility. The first is that properties, themselves, need to be able to determine how their behaviors might interact, so that they can potentially "negotiate" about ordering or customize their behavior in order to better interoperate with other properties. For instance, the versioning property might behave differently when attached to a document that also has a replication property. The second is that we need to be able to provide end-users with an understanding of the consequences of their actions. Since seemingly simple actions such as adding a property to a document might cause a variety of active properties to be executed, we need to be able to provide a generic framework in which the potential outcomes of actions can be determined. This also requires that we be able to determine something of the structure and behavior of active properties.

Our active properties are written in Java. As such, their "contents" – the code that they will execute – is largely inaccessible from user space, outside of the minimal structural properties that are available through the standard Java Reflection APIs. These are sufficient to be able to see what operations and what phases are being intercepted by the active property, but not to determine what that active property will do. Instead, a number of our applications are forced to depend on active property class names and "well-known" properties to be able to reason about the behavior of properties attached to documents.

One alternative would be to write active properties in a more declarative "little language" with less expressive power than full Java, and about which we might be able to reason more carefully. This approach has been used in other systems such as DPF [12]. We deemed this approach inappropriate for the initial explorations that Placeless Documents was designed to support; when we were engaged in the design we lacked sufficient experience with active properties to design such a language. The experience we have now gained suggests that the ability to use active properties to relate document behavior to the functionality of external services (e.g. external workflow engines or format conversion services) is an important feature of our design. In order to support this, we must necessarily give up strong control over the semantics of active property execution. A voluntary declarative specification of the side-effects or performance requirements of active properties may be incorporated in the future.

5. In fact, our implementation allows multiple instances of an active property to share code, so the overhead of multiple instances is not one of memory footprint, but one of actual code.

RELATED APPROACHES

There are two areas of related work relevant to our exploration of programming models for active documents. One is the set of investigations into active infrastructures of various sorts; the other is the exploration of compositional programming models for interactive behavior.

Active Infrastructures

Placeless Documents represents one approach to the provision of active infrastructures. Similar issues occur in other systems tackling similar problems.

The Bayou distributed database infrastructure gave programmers active control over the database system's policy for managing conflicts. By weakening the traditional ACID properties, replacing them with a more fluid set of "session guarantees" [23], Bayou provided a data storage layer more attuned to the needs of application domains such as cooperative work [9]. Bayou allowed programmers to attach pieces of code called "mergeprocs" to updates, which would be executed to resolve conflicts encountered as they filtered through the network of database servers. Like our active properties, mergeprocs were written in a full high-level language, raising the same sorts of introspection problems. However, since different mergeprocs would not be associated with the same update, Bayou had little need to support compositionality and so did not suffer the same problems of ordering, etc., that occur in Placeless Documents.

Active databases more generally provide a mechanism to incorporate dynamically computed data and "triggers" with data objects, allowing the data to respond actively to patterns of access [17]. These tend to be written in restricted languages, and the ways in which they can interoperate and rely on external services is much more restricted than we can offer, allowing active databases to offer stronger performance guarantees at the cost of expressiveness.

Active networking [27] is an approach to network architecture in which programs are executed inside the network architecture itself. This approach supports much more flexible management of network resources. For example, routers can exploit downloaded code to dynamically control routing patterns, allowing applications to specialize the network's response to their particular requirements. This is similar to work on composable protocols [16] in that it affects the configuration of networking behavior, but critically different in where it locates computation, and in particular that it allows computation to move around and to be configured dynamically. Work on Active Names [26] has explored inserting active mediation into just the name lookup process, so that dynamic or caller-dependent bindings are managed in a principled way.

Some interactive application toolkits, such as Prospero [4] and Intermezzo [8], have incorporated extensibility tech-

niques so that application developers can extend toolkit functionality or interfaces. However, they typically do not attempt to expose this extensibility model to end users as we do in Placeless Documents.

This is also true of explorations of active infrastructures in the operating systems domain. Work such as that on Spin [3], scheduler activations [2], exokernels [11] and the Mach External Pager [18] have all explored mediating the behavior of otherwise static infrastructure components. Much of the effort has been directed towards finding a balance between the expressiveness of the interface and the security and resource management implications of broadening it too far.

Compositional Interactive Behaviors

Unlike other approaches to active infrastructures, Placeless is focussed directly on interaction concerns. How does the active property model compare to other approaches providing compositional interactive behaviors?

In contrast to traditional "widget" programming, Myers' "interactors" model provides an encapsulation of interactive behavior that is separate from the graphical elements to which it is connected [15]. Interaction patterns and graphical elements of the user interface are developed and specified separately; interactors can then be attached to graphical elements to give those objects interactive behaviors. This is similar to the separation between documents and active properties; they also have similar compositional properties which make for similar programming experiences. However, Myers' interactors model is aimed specifically at the creation of graphical user interfaces, while the active property model is more general.

Active properties also resemble the programming model of prototype-based object-oriented programming languages like Self [25]. Like objects of a traditional OO language, Self objects combine data and activity (methods); but unlike the traditional approach, Self provides no classes to encapsulate object structures, but allows objects to inherit directly from each other, through a prototype mechanism. Placeless Documents is similar to Self in the way that users experience documents (objects) directly rather than in terms of predefined structures; and similarly, a number of our user interfaces are designed around the same principles of concreteness and uniformity that characterize Self user interfaces [20]. However, although our early designs incorporated it, Placeless Documents does not provide inheritance or the propagation of properties or activity through a "chain" of documents. This is because of Placeless' schizophrenic nature, being both an application infrastructure and a system for end users. While application developers might be expected to understand a model structured around prototype inheritance, we felt that end-user would not.

FOLLOW-ON WORK

The Placeless Document system is still under active development. In particular, a new "kernel" architecture (the core property storage and activation engine) is currently being developed, and explores more advanced ideas for the integration of Placeless's property mechanisms with the relational database beneath the covers.

Another feature that is being explored in this new implementation is a richer mechanism for application programmers to describe to the infrastructure the structure of their applications and their use of properties for data storage. In the first instance, this facility is aimed primarily at the use of static properties and schemas, but it also offers the opportunity to alleviate some of the problems associated with the composition of active properties. A declarative means for describing application needs allows the system to adapt itself to the needs of each application and to take a more active role in managing their interactions.

We have already described a number of ways in which the system has been changed to incorporate lessons learned from early application experiences (e.g. the introduction of client-side notifications). By working closely with application developers, we are still learning new ways to better match the conceptual model that Placeless offers to the needs of both programmers and end-users.

CONCLUSIONS

Infrastructures have traditionally been construed as passive elements of computing systems; activity has been concentrated at the end-points of the system, on servers or clients. However, researchers in a variety of systems arenas have been exploring the use of active infrastructures in order to specialise infrastructures to the needs of particular clients or to better capitalize on the increased performance of modern infrastructure components (e.g. increased network bandwidth). In the Placeless Documents project, we have been exploring an active infrastructure approach in the context of interactive document services and applications.

Active infrastructures have been conceptualized in many different ways, encapsulating many different programming models. The programming model for Placeless Documents has been designed to meet two goals. On one hand, it is powerful enough to allow application developers to create a wide range of applications that can take advantage of activity in the infrastructure. On the other, since we are aiming at *interactive* document applications, the programming approach must also encapsulate a conceptual model that is accessible to end-users. The Placeless Documents system blurs the traditional boundary between users and developers, since it gives users compositional control over the behavior of their document systems. While our approach is not as general as that of traditional end-user programming systems, it nonetheless requires that users be able to understand the encapsulation and composition model that our system offers.

Our conceptual model has been based on active properties. Static properties describe features of documents relevant to user needs; active properties add encapsulated code that affects how the document behaves. Properties can be controlled individually by end-users and application developers, can be composed to create complex behaviors, and provide a consistent interface for managing documents.

We have worked with a variety of people outside the development group to develop applications for Placeless Documents organized in terms of active properties. On the positive side, we have found that people take naturally to the

active property model, and can quickly and easily create applications factored into active properties. Active properties also encourage a decomposed model in which behaviors are persistently associated with documents so that application activity is distributed throughout the infrastructure and permanently available. On the negative side, we have encountered problems where active properties hide application behavior, making it hard for users to understand the consequences of their actions; document behavior may result in unexpected and unforeseen active property invocations. This has not caused any significant problems in the small applications we have explored, but raises some issues to be addressed in future explorations.

The active property model spans two worlds – the world of end-user document management and the world of application development. As computers become more powerful and applications need to be more and more radically adapted to the needs of different users and different environments, system developers will increasingly need to build bridges between these two worlds. In the Placeless Documents system, active properties have begun to provide us with some clues as to the problems and opportunities that these requirements will present.

ACKNOWLEDGEMENTS

We would like to thank Dirk Balfanz, Jacek Gwizdka, Minwen Ji, Eyal de Lara and Ian Smith for their bravery in experimenting with active properties.

REFERENCES

1. Abowd, G., Dey, A., Orr, R. and Brotherton, J. 1998. Context-awareness in Wearable and Ubiquitous Computing. Virtual Reality, 3, 200-211.
2. Anderson, T., Bershad, B., Lazowska, E. and Levy, H. 1992. Scheduler Activations: Effective Kernel Support for User-Level Management of Parallelism. ACM Trans. Computer Systems, 10(1), 53–79.
3. Bershad, B., Savage, S., Pardayk, P., Sirer, E., Fiuczynski, M., Becker, D., Chambers, C. and Eggers, S. (1995). Extensibility and Safety in the SPIN Operating Systenm. *Proc. ACM Symp. Operating System Principles* (Copper Mountain, CO). New York: ACM.
4. Dourish, P. 1998. Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications. *ACM Trans. Computer-Human Interaction*, 5(2), 109-155.
5. Dourish, P., Bentley, R., Jones, R. and MacLean, A. (1999). Getting Some Perspective: Using Process Descriptions to Index Document History. *Proc. ACM Conf. Supporting Group Work GROUP'99* (Phoenix, AZ). New York: ACM.
6. Dourish, P., Edwards, K., LaMarca, A. and Salisbury, M. (1999). Uniform Document Interaction with Document Properties. *Proc. ACM Symp. User Interface Software and Technology UIST'99* (Asheville, NC). New York: ACM.
7. Dourish, P., Edwards, K., LaMarca, A., Lamping, J., Petersen, K., Salisbury, M., Terry, D. and Thornton, J. (in press). Extending Document Management Systems With Per-User Active Properties. *ACM Trans. Information Systems*.
8. Edwards, K. 1996. *Coordination Infrastructure in Collaborative Systems*. PhD dissertation, College of Computing, Georgia Institute of Technology, Atlanta, GA.
9. Edwards, K., Mynatt, E., Petersen, K., Spreitzer, M., Terry, D., and Theimer, M. 1997. Designing and Implementing Asynchronous Collaborative Applications with Bayou. *Proc. ACM Symp. User Interface Software and Technology UIST'97* (Banff, Alberta). New York: ACM.
10. Edwards, K. and LaMarca, A. 1999. Balancing Generality and Specificity in Document Management Systems. *Proc. Seventh IFIP Conf. Human-Computer-Interaction Interact'99* (Edinburgh, Scotland).
11. Engler, D., Kaashoek, F. and O'Toole, J. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. *Proc. ACM Symp. Operating Systems Principles SOSP-95*, 251–266. New York: ACM.
12. Engler, D. and Kaashoek, F. 1996. DPF: Fst, Flexible Message Demultiplexing using Dynamic Code Generation. Proc. SIGCOMM'96. New York: ACM.
13. Fox, A., Gribble, S., Brewer, E. and Amir, E. 1996. Adapting to Network and Client Variation via Real-Time Distillation. *Proc. ACM Symp. Architectural Support for Programming Languages and Operating Systems ASPLOS-VII* (Boston, MA). New York: ACM.
14. LaMarca, A., Edwards, K., Dourish, P., Lamping, J., Smith, I. and Thornton, J. 1999. Taking the Work out of Workflow: Mechanisms for Document-Centered Collaboration. *Proc. European Conf. Computer-Supported Cooperative Work ECSCW'99* (Copenhagen, Denmark). Dordrecht: Kluwer.
15. Myers, B. 1990. A New Model for Handling Input. *ACM Trans. Information Systems*, 8(3), 289–320.
16. O'Malley, S. and Peterson, L. 1992. A Dynamic Network Architecture. *ACM Trans. Computing Systems*, 10(2), 110–143.
17. Paton, N. and Diaz, O. 1999. Active Database Systems. *ACM Computing Surveys*, 31(1), 63-106.
18. Rashid, R., Tevanian, A., Young, M, Golub, D., Baron, R., Black, D., Bolosky, W. and Chew, J. 1987. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *Proc. ACM Conf. Architectural Support for Programming Languages and Operating Systems* (Palo Alto, CA). New York: ACM.
19. Schilit, B., Adams, N. and Want, R. 1994. Context-aware Computing Applications. Proc. Workshop on Mobile Computing Systems and Applications (Santa Cruz, CA).
20. Smith, R., Maloney, J. and Ungar, D. 1995. The Self-4.0 User Interface: Manifesting a System-Wide Vision of Concreteness, Uniformity and Flexibility. *Proc. ACM Conf. Object-Oriented Programming Languages, Systems and Applications OOPSLA'95* (Austin, TX). New York: ACM.
21. Stein, L., Lieberman, H. and Ungar, D. 1987. A Shared View of Sharing: The Treaty of Orlando. In Kim and Lochovsky (eds), *Object-Oriented Concepts, Databases and Applications*. New York: ACM Press.
22. Swenson, K. 1998. Simple Workflow Access Protocol. IETF Internet Draft.
23. Terry, D., Demers, A., Petersen, K., Spreitzer, M., Theimer, M. and Welch, B. 1994. Session Guarantees for Weakly Consistent Replicated Data. *Proc. Intl. Conf. Parallel and Distributed Information Systems* (Austin, Texas).
24. Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M. and Hauser, C. 1995. Managing Update Conflicts in Bayou, A Weakly Connnected Replicated Storage System. Proc. ACM Symp. Operating Systems Principles SOSP'95 (Copper Mountain, CO). New York: ACM.
25. Ungar, D. and Smith, R. 1987. Self: The Power of Simplicity. *Proc. ACM Conf. Object-Oriented Programming Languages, Systems and Applications OOPSLA'87* (Orlando, FL). New York: ACM.
26. Vahdat, A., Dahlin, M., Anderson, T., and Aggarwal, A.1999. Active Names: Flexible Location and Transport of Wide-Area Resources. *Proc. 1999 USENIX Symposium on Internet Technologies and Systems (USITS).*
27. Wetherall, D. 1999. Active Network *Vision and Reality: Lessons from a Capsule-Based System. Proc. ACM Symp. Operating System Principles SOSP-17* (Liawah Island, SC). New York: ACM.