Xerox Palo Alto Research Center Technical Report CSL-01-1. April 20, 2001
http://www.parc.xerox.com/csl/

The Case for Recombinant Computing

W. Keith Edwards, Mark W. Newman, Jana Z. Sedivy

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304, USA
{kedwards, mnewman, sedivy}@parc.xerox.com

Abstract. Interoperability among a group of devices, applications, and services is typically predicated on those entities having some degree of prior knowledge of one another. In particular, they must be written to understand the type of thing with which they will interact, including the details of communication as well as semantic knowledge such as when and how to communicate. This paper presents a case for “recombinant computing”—a set of common interaction patterns that leverage mobile code to allow rich interactions among computational entities with only limited *a priori* knowledge of one another. We have been experimenting with a particular embodiment of these ideas, which we call Speakeasy. It is designed to support ad hoc, end user configurations of hardware and software, and provides patterns for data exchange, user control, and contextual awareness.

1 Introduction

Much of the focus of software support for ubiquitous computing has centered on isolated technologies, such as location sensing, multi-device user interfaces, ad hoc networking protocols and so on. While these are essential technologies, we propose that there are fundamental software architectural issues that are not being addressed by current research in ubiquitous computing.

If we postulate a world in which we will be surrounded by numerous devices and software services, this raises serious architectural questions. How will these devices be able to interact with each other? Must we standardize on a set of common “ubicomp” protocols? Must we agree on the semantics of every type of device or software that may be encountered in order to have fluid interoperability?

Currently, if I want to enable my desktop PC to print to a new printer, I have to install a driver for it. Perhaps more fundamentally, my desktop PC—and all the applications on it—first had to understand even that there *was* such a thing as a printer, and understand the semantics of such a device (the fact that documents can be printed to it, when it is appropriate to do so, and so on). Software and devices that do not understand the notion of a printer, and are not explicitly written to use such devices, will not be able to print.

This simple example points to what we believe is perhaps the fundamental architectural problem in ubiquitous computing: that current software infrastructures require that an application not only know about the things it will interact with, but also about the *types* of the things it will interact with.

In the future world of ubiquitous computing, it is unreasonable to expect that every device will have prior knowledge of every other type of device, software service, or application that it may encounter. Such knowledge would require anticipation of the semantics of such entities, and agreement on the means for communication with them. It is precisely the richness of the computational elements in such a world that prohibits us from building in *a priori* support for all such conceivable devices and software into all applications.

And yet, if our software infrastructures cannot easily accommodate interactions with previously unknown elements in the environment without being recompiled, upgraded, or replaced, ubiquitous computing will certainly not support the "calm" interactions envisioned by Mark Weiser [31]. It is more likely to be an endeavor fraught with the frustrations of software incompatibility, communication problems, version mismatches, and driver installations.

In the remainder of this paper, we present the case for an approach to software infrastructures called "Recombinant Computing." This approach points the way towards enabling fluid interoperation in ubiquitous computing by allowing entities to interoperate without having prior knowledge of each other. After presenting the essential features of this approach, we introduce the Speakeasy framework, which is a connection-oriented framework we have developed to explore the recombinant computing vision. We give an overview of this framework and discuss our initial experiences with it.

2 Network Effects

Note that the situation in computing is far different from, say, that of the telephone network. My rotary dial phone, circa 1975, still functions perfectly well and has never had an upgrade. Despite this lack of administration and maintenance on my part, I am able to place calls anywhere in the world. New functions, like call waiting, become available to me without me having to do anything other than request them. And I can transparently use this same device to connect to devices that were not even in existence when my phone was built, such as cellular telephones that exist on a variety of worldwide networks and use widely varying protocols.

The telephone system is perhaps the chief example of what has been called a "network effect" [6]. A network effect is a property that exists in a distributed system when a single component in the system increases in value as a result of actions elsewhere in the system. To put this in terms of the telephone example, each telephone in the network increases in value (utility) as other telephones come on line. This is because each telephone becomes capable of reaching more people, on more devices, without any additional work on the part of the device's owner. Network effects result because of combinatorial explosions of possible interconnections between the parts of the system.

Of course, there are architectural reasons why the telephone network has these properties, not least of which was a decision to place as much intelligence as possible in the network, rather than the handset. But another essential reason is that the telephone's interface to that network—a four wire RJ-11 connection (in the US), along with a simple signaling protocol to a central switch—is very standardized, and very narrow. By standardized we mean that the interface is ubiquitous; by narrow we mean that it is simple, easy to implement, and—once implemented—is capable of delivering the full range of the network's functionality to the connected device. There are numerous other technologies that exhibit network effects, including the fax, the web, and the iMode system in Japan, all of which have these same properties.

Our goal is to enable these same sorts of network effects in a much richer domain—arbitrary computational devices and software. We believe that the presence of such network effects is a *requirement* for making the ubiquitous computing vision practical. The user experience of ubiquitous computing is predicated on the ability to easily, safely, and fluidly interconnect and use arbitrary devices and software encountered in the environment. Architecturally, this experience depends on the ability of devices and software to connect with each other without requiring prior knowledge of each other. Such an architecture should provide the potential for a combinatorial explosion of possible interactions among devices and software, increasing the utility of each, and the total utility of the network.

3 Recombinant Computing

There are a number of approaches one might take to provide network effects for ubiquitous computing. For example, one might standardize on a common handful of protocols and device types into which all future work must fit. We believe such approaches may be overly restrictive for the domain of arbitrary computational devices and services, however.

Another category of approaches, which we have termed *recombinant computing*, allows the dynamic extension of computational entities through the use of mobile code. We believe that such approaches can provide greater flexibility and power than, say, agreement on a simple, static set of communication protocols. We describe our work in terms of a set of "patterns" (to use the term loosely) that can allow a rich range of interactions among arbitrary computational entities, particularly when coupled with the presence of a user to provide semantic interpretations to given entities.

Our vision of recombinant computing is built atop previous work in software architecture. In particular, we use a component model in which each computational entity on the network is treated as a separate component. The presence of new components is detected through the use of dynamic discovery protocols, and we rely upon a mobile code framework to deliver the implementations of components needed at runtime. The next sections describe each of these foundation technologies, and motivate our discussion of recombinant computing by illustrating why these technologies are necessary but, in themselves, insufficient to support the network effects that must be our goal.

3.1 Component Frameworks

Component frameworks such as JavaBeans [26] and COM [14] provide a very simple form of recombination: they allow developers to use code that they did not themselves write, and thus promote code reuse and modularity.

As an example, a word processing application such as MS Word does not itself provide built in support for every printer on the market. Instead, printing functionality is isolated into a component (a driver), which is abstracted from the rest of the Word functionality. This same driver is reused by the other applications on the operating system that need to print, and isolates Word from having to understand the particulars of an individual printer.

Traditional component frameworks, however, are insufficient to enable arbitrary software interconnection because they require the user of a component to have explicit prior knowledge of not only the *interface* provided by that component, but also its *semantics*. The users of such frameworks are application developers who use their understanding of the specialized interfaces as well as the semantics of use to create appropriate interconnections among components. This “hard wiring” of entities does not enable the sorts of arbitrary dynamic composition that produce network effects. Furthermore, they typically support only static (compile time) associations among components. This last limitation, however, can be overcome by the next essential feature of recombinant computing, discovery.

3.2 Discovery Systems

Discovery allows applications to be notified when new components are needed or available, so that they can dynamically install components they are already equipped to use. A very simple, non-networked form of discovery is Microsoft’s Plug and Play, or PnP [18]. PnP can inform applications and the operating system when a new software component is needed. So, for example, if a user plugs a new printer into her computer, and the driver software component that supports that printer is available on the system, then it will be loaded and used. In essence, the system itself takes responsibility for deciding when new components are necessary to allow applications to function. Networked versions of discovery (such as the Simple Service Discovery Protocol portion of Microsoft’s Universal Plug and Play [10], UDDI [28], and Salutation [23]) extend this ability to the network; they can discover services and devices that exist on remote hosts, as opposed to only those services and devices installed locally.

Such discovery systems increase the utility of component frameworks, since particular components can be automatically selected and bound at runtime, rather than at compile time. But, in themselves, they still fall short of supporting arbitrary, fluid interconnections among devices and software. To return to the Word printing example, PnP may tell the system that a new printer has been installed and that a driver is needed to support it. But if that driver component is not available locally, then the binding between application and component cannot be established, and explicit human intervention—putting the component on a floppy and installing it manually—is required. So while discovery allows the system to have some degree of automatic adaptability to new situations, it is insufficient to allow arbitrary recombination.

3.3 Mobile Code

The requirement that software components used to access network resources be pre-installed is overcome through mobile code-based systems. Mobile code is the ability to move executable code across the network, to the place where it is needed, and then execute that code securely and reliably. The Java runtime environment is the best-known example of a system in which mobile code is used extensively, to support applets, for example, and distributed polymorphism in the case of RMI [32]. Mobile code is also possible using the bytecode language runtime which supports the languages in Microsoft's .NET platform, such as C# [20].

Currently, Jini [30] is the only widely available platform that combines a component framework with its own dynamic, network discovery protocols, and mobile code. However, while platforms such as this address a real need, and can provide significant benefits to users, they are not sufficient in themselves to enable the vision of ubiquitous computing.

A mobile code-based architecture would enable Word to not only detect that a new printer is available, but determine what code is needed to use it, and then deliver that needed code *on demand*. This feature is one of the chief claimed benefits of Jini, that services (such as printers) can be discovered by clients dynamically, that services carry with them the code needed to use them, and that this code is dynamically downloaded to clients as needed.

As powerful as this solution is, however, it doesn't allow Word to do new things *other* than print. It does not, for instance, allow the application to suddenly scan, or synchronize with a PDA, or allow collaborative co-authoring. In short, while it can provide a new way for Word to do things it already knows how to do, it cannot allow Word to do new things it wasn't explicitly written to do. A Jini-enhanced version of Word would still need to be written against a specific and predefined API for printing.

4 Recombinant Interfaces

Powerful as the approaches illustrated above are, the problem is that all of these very specific interfaces must be understood by the authors of the applications that will use them. Applications must be specifically written to use every domain-specific component interface (printing, file storage, image capture, and so on), meaning that new interface types that appear in the future can only be supported by either (1) retrofitting them into some other, previous interface, or (2) rewriting the application to accommodate the new interface. We believe that it is impractical and implausible that applications can be written to have explicit prior knowledge of the full range of components that they may encounter.

This is the chief obstacle to enabling network effects—the lack of a small, *fixed* set of interfaces that are both standardized (meaning that applications can expect them to be supported by all components they may encounter) and narrow (meaning that they can reasonably be supported by all applications, regardless of their semantics, and without exorbitant work), and yet are expressive enough to capture a wide set of the functionality of arbitrary devices and programs.

Our solution is to develop a set of *recombinant interfaces*, which are programmatic interfaces that specify the ways in which components interact with one another. These are “trans domain” interfaces that describe the ways in which components can extend one another's behavior at runtime, rather than describing the domain-specific functionality of the components (such as printing or file storage).

These interfaces establish the groundwork, or minimal agreed-upon language of interaction, that all components are expected to support in order to be able to interact with one another. In essence, they define the common syntax of interaction among components. These interfaces do not express the domain-specific semantics of the component, however. As we discuss next, much of the semantic understanding of when and why components should interact can and should be provided by users.

5 The Importance of User-Provided Semantics

Fundamental to the vision of recombinant computing is the belief that it is *users* who will provide the semantic understanding of what components actually do. In other words, recombinant interfaces define *how* components interact with one another, and reveal information designed to afford humans the ability to decide *when* and *whether* such components *should* interact.

For example, a PDA that knows nothing about printers would be able to print, because aspects of the behavior of printers are represented using these interfaces. Of course, without knowing the semantics of “printing,” a PDA would never simply use a component that it encounters—it would not know whether that component printed the data that was sent to it, stored it, or simply threw it away. Instead, the device would most likely inform the user that a component calling itself a “printer” had been found, and leave it to the user to make the decision about when and whether to use this device.

Put another way, the programmatic interfaces define the syntax of interaction, while leaving it to humans to impose semantics. It is the stabilization of the programmatic interfaces that enable network effects; humans can, presumably, “understand” new components in their environment, and when and whether to use them, much more easily than machines can (and, it should be noted, without the need to be recompiled). This semantic arbitration happens at the user level, rather than at the developer level as in traditional component frameworks.

Our emphasis on keeping the user “in the loop” is based on the recognition that in order to enable network effects to take hold, it will be necessary to allow users to carry out operations for which no application has yet been developed. Typical ubiquitous computing environments will be profoundly heterogeneous and ad hoc in their composition, and it is infeasible to expect application developers to keep pace with the evolution of these environments and with the things that people will want to do with them.

It may be possible to offload some aspects of semantic interpretation from users to other computational entities in order to make some aspects of recombination more fluid. Machine learning techniques might be applied to individual users' patterns of use, for example, in order to predict what types of configurations a user would be likely to attempt in the future. While such techniques could add significant value to a recombinant infrastructure such as the one we propose, we do not think it advisable to rely on the success of such techniques. Even the most “intelligent” infrastructures of this type would likely have to defer to users on some occasions, such as when they do not have enough information to make a decision. By focusing on the basic syntax of interaction and separating out the semantics, the recombinant computing approach enables user-driven interaction among arbitrary computing elements in the near term and leaves room for growth in the domain of making such interactions more intelligent.

Of course, there will still be applications for which an application developer *has* foreseen a need and for which a richer, domain-specific interaction between components will be desirable. Note that nothing in our description of recombinant computing rules out the possibility that components will also implement domain-specific interfaces. To paraphrase Freud, sometimes a printer is just a printer. By implementing a set of recombinant interfaces *in addition to* its native printer interface, the printer component can increase its value as well as the value of all of the other recombinant components with which it can now communicate.

Thus far, we have presented the notion of recombinant computing and introduced the key features of this approach. We have argued that recombinant computing is the right approach for realizing network effects in ubiquitous computing. In the remainder of this paper we present a specific realization of the recombinant computing ideas. The next sections describe the current set of interfaces that make up the Speakeasy framework, and the rationale for each. Afterwards, we examine the “applications”—configurations of recombinant components—that we have built using these interfaces.

6 Speakeasy: A Recombinant Computing Framework

We have created an initial architecture, dubbed *Speakeasy*, to explore the models of recombinant computing outlined above. While there may be many possible ways to embody recombinant functionality in a set of programmatic interfaces, Speakeasy focused on one such approach, which might be termed “connection oriented.” In the Speakeasy interfaces, components are first and foremost viewed in terms of their ability to produce or receive information. Secondary interfaces exist to enable sensemaking, navigation of the world, interaction, and so forth, but our primary idiom of recombination is the notion of a connection between components.

Broadly, our interfaces define three general categories of functionality that components may support:

- Connection. How do components exchange information with each other?
- Context. How do components reveal information about themselves?
- Control. How do components allow users (and other components) to effect change in them?

Any given component may support multiple of these interfaces. For example, a given component may be able to exchange information with other components and may also reveal information about itself.

Our contention is that applications written to understand this handful of interfaces will be able to interact with *any* component that may come along in the future that exposes its functionality in these terms. In other words, such a software system will be able to use new components that appear in its environment, without recompilation or update.

Certainly, one could make the claim that this approach trades one problem for another. That, rather than writing applications to specific printer, filesharing, etc., interfaces, we're requiring them to write to simply another set of interfaces. While this is true, we believe that a thoughtful set of interfaces that can describe functionality independent of any particular domain, coupled with the ability to transmit and execute mobile code, can allow applications far greater "coverage" of functionality than has been possible before.

In our choice of interfaces, we have attempted to define the smallest set of abstractions that are both necessary and sufficient to support arbitrary connections between arbitrary components. For our starting point, we take the fact that, at its most primitive level, the ability to move data between entities is crucial. The notion that data can be shared among entities, and moved among them, is implicit in the very idea of recombination. However, the ability to simply move data around is insufficient without some control mechanism to initiate the transfer and make decisions about when this should be done. For this reason, our interfaces include explicit models for both Connection and Control.

However, these two abstractions on their own are still somewhat limiting. Whether or not to move data between entities will likely be a complex decision requiring specialized information (such as location or administrative domain) about each component involved. We make the claim that an appropriate generic representation of this specialized component information is to encapsulate it as attributes of an extensible "Context" object that is contained within every component. Each component implements a common interface for retrieving and modifying its context, thereby providing the user (or some inference engine) with information it will need to make decisions about how to use the component.

6.1 Connection

As mentioned previously, the fundamental metaphor in our approach to recombinant computing is *connection*. A connection between two components indicates an association for the purpose of transferring data. Such a transfer may represent a PDA sending data to a printer, a whiteboard capture camera storing a snapshot in a filesystem, or a laptop computer sending its display to a networked video projector.

Each of these different senders of data—a PDA, a camera, and a laptop computer—may use very different mechanisms for transferring their data. A sender of high-resolution video, for example, is likely to use a streaming protocol that is adaptive to the transport over which it is being run, perhaps by increasing compression or lossiness over slow media. A PDA sending a file to a printer will likely use a very different data transfer mechanism and indeed, a lossy connection such as that used by video will be unworkable for this interaction.

This example points out the infeasibility of deciding on a handful of data exchange protocols that all components agree upon. Each component may have its own semantics with regard to sending data, and these semantics are likely to be reflected in the protocols used. Therefore, it is infeasible to build in support for all of these potential protocols “up front.”

Our approach is to use a pattern whereby a sender of data can extend the behavior of its receiver to enable it to transfer the data using a desired protocol. Rather than providing the data directly, a sender transfers a *source-provided endpoint* to its receiver. This is a custom, sender-provided communication handler that lives in the receiver. Different senders will export different endpoint implementations, and the code for these different implementations will be dynamically loaded by receivers as needed.

This arrangement gives the sender control over both endpoints of the communication; the sender can choose an appropriate protocol for data exchange, without the need for these protocols to be built into every receiver.

Once the endpoint code has been received, the actual transfer of data is initiated. The endpoint communicates with the remote sending component, using whatever protocols have been dictated by the creator of that component. This data is then returned to the receiver as a stream of bytes in the desired format. The behavior of the receiver at this point depends on its particular semantics—a printer will print the data it receives, while a file system will store the data as a file.

Our current interfaces allow components to be either senders or receivers of data, or both. Both senders and receivers can indicate the types of data that they can handle; we currently use standard MIME [2] type names. Programs and users can use these types to select components that are capable of exchanging data with one another.

Figure 1 illustrates the operation. In the top part of the figure the receiver initiates a connection with a sender of video data, and the sender returns an endpoint to the caller (in this case, an endpoint capable of using a streaming protocol for MPEG2 data). This portion of the operation occurs using the “public” connection interfaces. After this, data is transferred to the receiver via the endpoint through a “private” protocol between the endpoint and the source.

Design Discussion. Several design decisions are represented in this arrangement. First, the same mechanisms are used to transfer “one shot” data (such as a file) as are used to establish long-lived connections. All receivers read data from a stream provided by the endpoint, until an end-of-data marker is reached. We felt that this was a desirable choice, since all receivers must be prepared to deal with long-lived connections anyway, either because of slow connection speed, or simply because the sender-provided endpoint transfers data in a streaming format.

A second design choice is that the interfaces presented here allow any party to initiate the transfer. For example, a third party (such as a “browser” application, whose function is similar to that of a file browser, but provides access to components) can fetch an endpoint from a sender and provide it to a receiver. Such an arrangement would cause the receiver to read directly from the sender, using the sender’s endpoint implementation (as provided to it by the browser). A receiver could also be written to automatically initiate connections with certain senders, if appropriate.

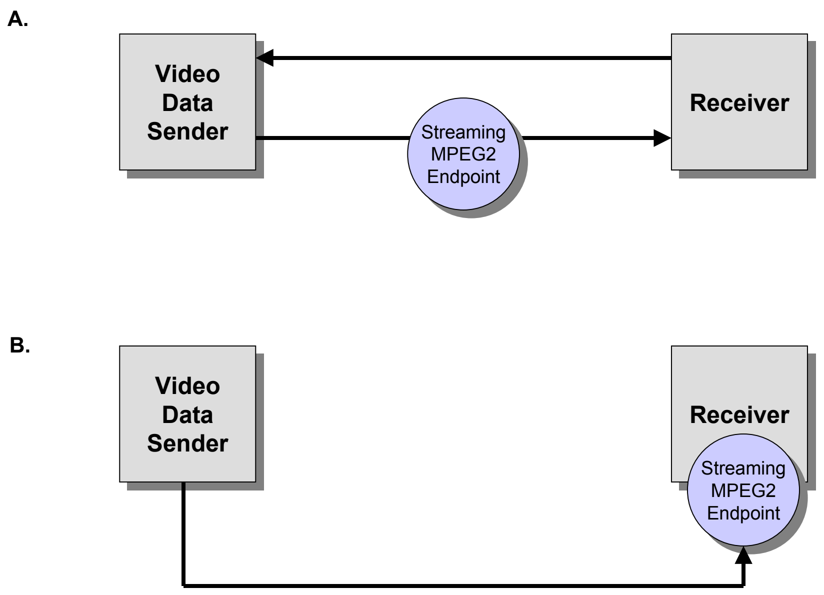

Figure 1 Data transfers in Speakeasy use source-provided endpoint code. In A, a receiver initiates a connection, which causes a source-specific endpoint to be returned. In B, this endpoint is used to fetch data from the source using a source-private protocol.

Endpoints do not remain valid indefinitely. Any given sender may, potentially, be engaged in any number of transfers with receivers. Senders and receivers must have a way of cleaning up after failures of either. For this reason, endpoints are *leased* from the senders which granted them. Leasing is a mechanism by which access to resources is granted for some period of time; to extend this duration, a continued proof-of-interest is required on the part of the party using the resource [11]. One benefit of leasing is that, should communication fail, both parties know that the communication is terminated once the lease expires, without any need for further communication.

The Tyranny of Types. A clear limitation of this approach is that, even though all data exchange uses a common interface that supports arbitrary, specialized protocols for transfer, many parties will have to have some semantic knowledge of the types they will exchange. For example, a printer might only accept Postscript data, while a web camera might only be capable of producing JPEG images. These two components would not be directly capable of exchanging information, because they are not data type compatible, even though they would speak compatible interfaces.

Agreement on type information is not a requirement for all interactions. A filesystem, for example, may happily store any type of data it receives in a file, without the requirement that it have any particular knowledge of those data types. But many components, including any that interpret or act on the *content* of the data they receive, will be able to process and handle only certain data types.

Although this need for type agreement limits the universality of the data connection interfaces, the interfaces do still provide a significant benefit, namely, the ability for any two components that *can* agree on types (or, alternatively, that don't *care* about types) to exchange information dynamically. Furthermore, the recombinant nature of components in our environment makes it easy to deploy, discover, and use data transformation services that can mediate between components that wish to share data, such as Ockerbloom's mediators [22] or Kiciman and Fox's paths [15].

6.2 Context

Simply knowing that a component can be a sender or receiver of data provides very little utility if there is no other way to find out more information about that component. Thus, all Speakeasy components are able to provide *context* about themselves. Context is simply a representation of a set of attributes that might be considered salient for a given component: its name, location, administrative domain, owner, version information, and so on. A user, through an application such as a browser, would be able to organize the set of available components based on location, owner, and so on.

Context is not only useful for making sense of the set of components available in the environment, but can also be used to mediate and control the behavior of individual components. For example, a video projector may allow any co-located user to use it directly, without further access control. The same projector, however, may require the approval of a co-located user when a remote user attempt to access it. Other components may similarly adapt their own behavior based on these attributes.

To support both of these uses, Speakeasy uses two contextual interfaces. First, all components can reveal information about their own context when asked. Users or applications can organize and understand the set of available components using this information. Second, all operations in Speakeasy require that the caller provide its own context. For example, to acquire an endpoint from a sender, the initiator of a transfer must provide the sender with its own context. This mechanism allows components to adapt their behavior to their users. A fileserver in the Xerox administrative domain, for example, would likely only interact with components in the same domain.

Currently, our representations of context are very simple; this is an area of ongoing work for us. The actual contextual data that is exchanged is a remote proxy object that communicates to the component to which it belongs. This proxy object reveals a simple map of key-value pairs, with names indicating contextual attributes (such as "Name," "Location," and so on), and values that are arbitrary objects, and may include digitally signed data such as certificates.

The set of valid keys is extensible, as we do not believe any fixed organization is likely to support the contextual needs of all applications or components. We do believe, however, that certain keys are likely to be commonly understood and used across components.

The basic model here—that components provide their own context—affords decentralization, since the interfaces do not require the presence of some centralized, always-available resource. The design does not preclude the presence of a different context infrastructure "underneath" the contextual APIs, however. So particular components could internally use sensing and aggregation infrastructures such as the Context Toolkit [4], or context storage and notification infrastructures such as Intermezzo [7], as long as these implementation artifacts are not exposed to clients.

An alternative approach for exposing this kind of information is to provide accessor methods (such as getLocation(), getOwner(), and so on) on the component through specialized programmatic interfaces. However, this is precisely the kind of specialization of interfaces that we hope to avoid: since interfaces are static, it is unworkable to decide upon a fixed set of accessors for a fixed set of information that we hope is applicable to all components. Exposing such information through specific accessors on the component provides neither extensibility to new information nor a separation between the semantics of such information and the interface used to access it.

Our design allows for the possibility of rich, complex interactions between components if the entities have some *a priori* knowledge about each other, but at the same time permits simple, general interactions in the absence of such knowledge. Components and applications can agree on the syntax of interaction (the interface for retrieving context) without having to have total agreement on the semantics of the various contextual attributes themselves. Applications and users are free to use the aspects of context that are salient to them (and understood by them) while ignoring others.

6.3 Control

There are, of course, many aspects of component behavior that are orthogonal to establishing data connections. In the example of a printer component, for example, the likely reaction of a printer to receiving data is to print it. Although the simple connection interfaces provide the minimum functionality that must be in place to send data to a printer, they cannot capture the full range of functionality of such a device. There is no notion of full duplex, or color versus black and white, or stapled output in the connection interfaces, for example. Nor should there be, since these are clearly concepts specific to printers.

How, then, can we provide access to such component-specific notions without requiring clients to potentially understand *all* such details?

Our approach is to provide users with mechanisms for acquiring user interfaces to components, and to provide applications with mechanisms for altering component state based on user interaction with those UIs. These two techniques are in keeping with our philosophy that the programmatic interfaces must provide only the bare mechanisms necessary for interaction, along with the means to defer to the user when appropriate. These techniques are discussed below.

Component-Specific User Interfaces. Any Speakeasy component can provide one or more complete user interfaces, at the request of a caller (and, like all other operations, requesting a user interface requires that the caller establish itself with the component by providing its context). These user interfaces are downloaded on demand by callers where they can be presented to users. Applications need not have built-in support for explicitly controlling any component.

Applications that need to display the UI for a component can select from potentially any number of UIs associated with a given component, by specifying the requirements of the desired UI. For example, a browser application running on a laptop computer might request a UI that requires and uses a full-blown GUI toolkit already present on the machine, while a web-based browser might request HTML or XML based UIs.

This approach is flexible in that it allows components to present arbitrary controls to users, and in that it allows multiple UIs, perhaps specialized for different classes of devices, to be associated with a given component. The primary drawback is that it requires each component writer to create a separate UI for each type of device that may be used to present the interface. A possible solution would be to use some device-independent representation of an interface, such as those proposed by Hodes [13] or the UIML standard [12], and then construct a client-specific instantiation of that UI at runtime. We are not currently focusing on developing such representations ourselves, but rather on the infrastructure that would be used to deliver such representations to clients.

Persistent Configuration of State. Under this model, the programmatic interfaces provide the tools for accessing component-specific UIs, without the need that applications understand any component-specific functionality. One additional interface we felt was required, however, was the ability to make a set of user-supplied configurations persistent.

For example, suppose that a user always wishes to get full-duplex, color, stapled printouts from a given printer. Clearly, having to display the UI for the printer and configure each of these settings every time the printer is used would be a great inconvenience. To prevent such hassle, we need a way for an application such as a browser to recreate a particular configuration of settings in a component, but again without requiring that the application understand any component-specific concepts.

Our approach is to allow applications to request that any component provide it with a representation of its state. This representation—which can essentially be treated as an opaque token—is a contract between the component and its client that the component will attempt to recreate the state represented by the token if the client provides it to the component at a later time.

Figure 2 illustrates this model. Here, a browser application interacts with a sender of video data. The browser requests the user interface for the component, perhaps at the direction of a user. Once the browser receives the UI and the user interacts with it, the UI transmits new configuration information back to the remote component, using a private protocol decided upon by the component's builder. When the interaction is finished, the browser may request a state token from the component, which persistently represents the particular parameterization of the component effected by the UI. In the future, the browser can recover this particular parameterization by returning the token to the component without having to have specific knowledge of the parameters themselves and without having to redisplay the UI to the user.

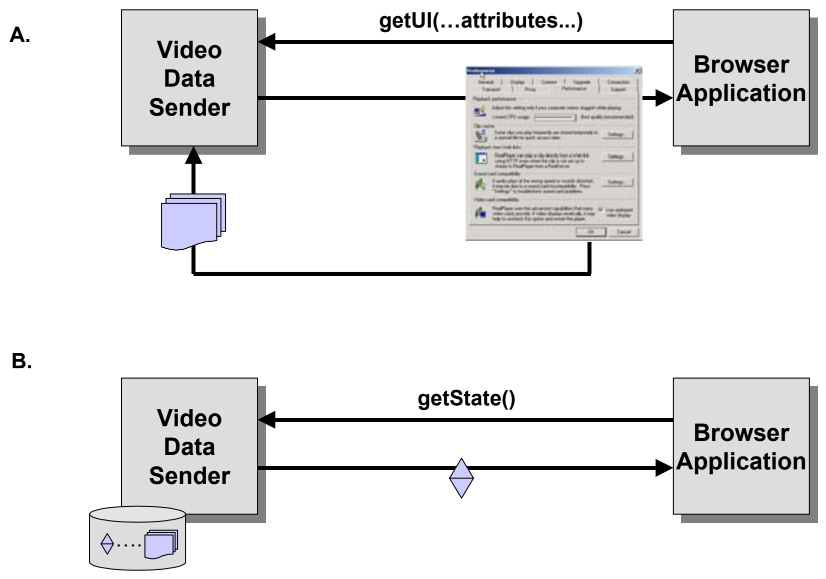

Figure 2 Control operations allow configurations of components, and recovery of configurations. In A, a browser requests a UI; as the user interacts with the UI, the component's configuration is changed via a component-private protocol. In B, the browser fetches a state token that can be used to recreate this configuration in the future.

7 Components and Applications

To date we have built approximately a dozen components, some of which represent fairly complex behavior, as a means of refining our interfaces and exploring useful combinations of functionality. For example, we have a video display component capable of accepting digital video input from a source such as a laptop. This component is typically embodied as a network-accessible digital projector. Unlike a typical projector, which requires a direct, physical VGA connection, this component exports our data receiver interface and can be accessed and used by any entity on the network capable of producing a compatible data stream.

We have also built a whiteboard mosaic and image capture component, based on the ZombieBoard technology [24]; a digital camera component based on the CamWorks technology [21]; a filesystem component based on the Harland/Placeless technology [5]; a Powerpoint viewer component that can render a Powerpoint presentation as a slide show, and provide controls for that slide show; a screen capture component that can redirect a computer's standard monitor display to another component (such as a projector); and a number of others.

Currently, these components are situated in work areas and offices around PARC. Users access and control them through a number of browser-style applications we have built. These browsers allow connections to be established between arbitrary components of compatible data types, control over devices, and exploration of the space of available components.

Even with a fairly small number of components, there are a number of configurations possible, which provide various applications of the Speakeasy framework. The interfaces and programming patterns we have developed are sufficient to allow presentations without a laptop, for example. Using a browser interface running on a PDA, a user would interconnect three separate components. First, the user interacts with his or her “home” filesystem component to find a file containing the desired presentation. This file is then connected to a Powerpoint viewer component, typically running on the user’s desktop PC. This component is capable of producing a data stream compatible with our display component, which is typically embodied as a small, network-connected computer attached to a projector or flat panel display in a conference room. The Powerpoint viewer is then connected to the display to complete the chain.

Control over the presentation is accomplished through the interfaces presented by the various components. The projector component presents a UI allowing the user to turn it on or off, for instance; the filesystem component allows the user to select a desired file. During the actual presentation, the user will likely interact only with the UI of the Powerpoint viewer component, which allows slideshow-style controls over the presentation. This control UI is separate from the data connection, and can be run on a device such as a PDA while the presentation is ongoing.

During the presentation, the user can interact with other components. For example, the user can browse and find a whiteboard capture component, and cause it to capture a whiteboard image and display it using the projector, or save it in any reachable filesystem.

8 Implementation

As mentioned earlier, Speakeasy is a set of interfaces designed to leverage mobile code, discovery, and component frameworks.[1] As such, we have chosen to focus on the interfaces themselves and reuse existing technologies where appropriate.

In our current implementation, we use Java’s facilities as our mechanism for the secure transfer and execution of mobile code. This means that, in our prototype, all components must be represented by proxy objects expressed in Java bytecodes.[2] We use the Jini multicast and unicast discovery protocols as our discovery layer [27], and Java in-

1. Speakeasy also provides an infrastructure to support building and using components, of course. For example, our implementation supports checkpointing and recovery of component state, code libraries to remotely log, monitor, and administer components, and so on. These details are not covered in this paper.

2. This does *not* mean, however, that components themselves must be written in Java. Their proxies can communicate with “native” components written in any language.

terfaces as our interface description language. Our control mechanisms build atop the ServiceUI framework [29] for associating user interfaces with Jini services.

We do not believe that there are any artifacts in the Speakeasy code that prevent other technologies from being used as a base, as long as they provide mechanisms for describing the interfaces to components, discovering the existence of those components, and then dynamically providing the code necessary to use them. The Microsoft .NET platform, for example, would likely be able to support our recombinant interfaces. Alternative discovery protocols—such as those based on the Bluetooth Service Discovery Protocol (SDP) [1], for instance—would clearly provide benefits.

The set of components described in this paper have all been implemented using our framework, and interact with one another solely using the recombinant interfaces described here (that is, they do not resort to other, functionality-specific interfaces). We have developed a pair of client browsers, including a “raw” browser used for debugging, and a webserver-based client that allows access to component functionality from any platform with a web browser (for example, a wirelessly connected PDA). We have also developed scenarios for a number of other client applications.

9 Related Work

Section three of this paper presented a number of technologies atop which our model of recombinant computing builds. There are, however, a number of other relevant technologies, including other efforts in the research and commercial community focused on top-to-bottom infrastructure approaches to ubiquitous computing

First, our efforts are loosely inspired by work in programming language patterns [9] and meta-object protocols [16]. In particular, the model of software and hardware components that can alter the environments in which they exist is reminiscent of work by Kiczales, et al., on meta-object protocols, which are mechanisms by which applications can introspect and modify the behavior of their own runtime environments. Our work extends this notion of reflection and adaptability across network boundaries.

A number of approaches, including Universal Plug and Play [19], and the .NET combination of UDDI for discovery [28], WSDL for interface definition [3], and SOAP for communication [25], are examples of what might be termed “service bus” models of interaction. These systems provide a platform on top of which components can discover and communicate with one another. But they still require components to have knowledge of the interfaces and semantics of the components they will interact with.

The CoolTown project at HP labs [17] is an example of an infrastructure for pervasive computing. Building on the ubiquity of web technology, the CoolTown approach provides access to all entities through URIs and handles communication through HTTP and HTML. While this is a certainly a promising approach, web technology evolved primarily as a set of document sharing protocols and hence imposes some inherent limitations on the kinds of interaction possible within its confines. We believe that by specifically designing interfaces towards distributed computation and user control, we can explore a richer set of interactions.

The CoolTown project also uses a “physical presence” model of service discovery, which requires the user to be physically near the devices they want to interact with. While this is an intuitive and straightforward approach with an easy to understand security model, we feel that it is important to not build this restriction into the infrastructure. There are many kinds of services for which physical location is not an important attribute (a format conversion service for example) and which the infrastructure for a generic distributed component network should be able to support.

The iRoom project also integrates diverse devices and software services [8]. Communication between entities in this environment works via an Event Heap which all components can query and send events to. This model requires that the source of an event must understand the format and protocol required by the intended recipient. This approach works very well in an environment such as the iRoom where system administrators carefully add each new entity and developers understand the programmatic interfaces of the elements already in the room. However, we believe that it does not scale appropriately in an environment where new entities have no prior knowledge of their peers on the network.

10 Conclusions and Future Directions

The primary goal of this paper is to introduce recombination as an essential problem that must be overcome for the ubiquitous computing vision to become a reality. The secondary goal is to begin a discussion on approaches to recombination.

The Speakeasy project is focusing on a connection-oriented approach to recombination, in which components interact with one another largely through transfers of information. To date, we have focused on developing an exploratory set of patterns that leverage mobile code to allow arbitrary computational entities to find, control, and interact with one another without prior knowledge. While these patterns have been refined through experience, and we have confidence that they work well for the applications illustrated above, we firmly believe that other approaches to recombination are possible. Such other approaches may well have other affordances, and may support different applications than the ones presented here (and, indeed, may support the applications shown here better).

In the near term, our project has two goals. First, we plan to continue to refine our recombinant interfaces through experimentation, component building, application building, and use. We are already experimenting with a number of mobile code-based strategies to eliminate the “tyranny of types” problems present in our current interfaces, and believe that we can extend the range and power of these interfaces. We are also investigating techniques for transparently extending the ability of components and applications to negotiate connections over arbitrary network interfaces.

Second, we wish to begin an exploration of issues of policy, user experience, and security in a recombinant world—just because arbitrary components have the ability to interact with one another doesn’t mean that they *should* interact with one another. How we will effectively understand and use the world around us in such a setting is a key issue we are beginning to look at in parallel with our ongoing focus on low-level interconnectivity and control issues.

References

[1] Bluetooth Consortium (2001). *Specification of the Bluetooth System, version 1.1 core*. http://www.bluetooth.com. February 22, 2001.

[2] Borenstein, N., and Freed, N. (1992). "MIME (Multipurpose Intermet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Messages." Internet RFC 1341, June 1992.

[3] Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). *Web Services Description Language (WSDL) 1.1*. http://msdn.microsoft.com/xml/general/wsdl.asp. January 23, 2001.

[4] Dey, A.K., Salber, D., Abowd, G.D. (2001) "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," to appear in *Human Computer Interaction Journal*, vol. 16, 2001.

[5] Dourish, P., Edwards, W.K., LaMarca, A., Lamping, J., Petersen, K., Salisbury, M., Thornton, J., and Terry, D.B. (2000). "Extending Document Management Systems with Active Properties," *ACM Transactions on Information Systems (TOIS)*, 2000.

[6] Economides, N. (1996). "The Economics of Networks," *International Journal of Industrial Organization*, 14:2, March, 1996.

[7] Edwards, W.K. (1994). "Session Management for Collaborative Applications," Proceedings of *ACM Conference on Computer-Supported Cooperative Work (CSCW)*, October, 1994. Chapel Hill, NC.

[8] Fox, A., Johanson, B., Hanrahan, P., Winograd, T. (2000). "Integrating Information Appliances into an Interactive Space," I*EEE Computer Graphics and Applications* 20:3 (May/June, 2000), 54-65.

[9] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, ISBN 0201633612, January 15, 1995.

[10] Goland, Y.Y., Cai, T., Leach, P., Gu, Y., Albright, S. (1999). *Simple Service Discovery Protocol/1.0: Operating Without an Arbiter*. Internet Engineering Task Force Internet Draft. http://www.upnp.org/draft_cai_ssdp_v1_03.txt.

[11] Gray, C.G., Cheriton, D.R. (1989). "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency," *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 202-210, December, 1989.

[12] Harmonia, Inc. (2000). *User Interface Modelling Language 2.0 Draft Specification*, http://www.uiml.org/specs/uiml2/index.htm.

[13] Hodes, T., and Katz, R.H. (1999). "A Document-Based Framework for Internet Application Control," *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS)*, Boulder, CO, October 1999, pp. 59-70.

[14] Iseminger, D. (2000). *COM+ Developer's Reference*. Microsoft Press. ISBN 0735611386. September, 2000.

[15] Kiciman, E., and Fox, A. (2000). "Using Dynamic Mediation to Integrate COTS Entities in a Ubiquitous Computing Environment," *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing (HUC)*, September, 2000.

[16] Kiczales, G., Des Rivieres, J., Bobrow, D. (1991). *The Art of the Metaobject Protocol*. MIT Press, ISBN 0262610744, September 1991.

[17] Kindberg, T., and Barton, J. (2000) “A Web-Based Nomadic Computing System,” HP Labs Technical Report HPL-2000-110. http://cooltown.hp.com/papers/nomadic/nomadic.htm, 2000.

[18] Microsoft Corp. (1999). *Plug and Play Specifications*, http://www.microsoft.com/hwdev/respec/pnpspecs.htm, April 29. 1999.

[19] Microsoft Corp. (2000). *Universal Plug and Play*, http://msdn.microsoft.com/library/psdk/upnp/upnpport_6zz9.htm. December 5, 2000.

[20] Microsoft Corp. (2001). *The C# Language Specification*. April 25, 2001, Microsoft Press.

[21] Newman, W., Dance, C., Taylor, A., Taylor, S., Taylor, M., Aldhous, T. (1999) “CamWorks: A Video-based Tool for Efficient Capture from Paper Source Documents,” *Proceedings of the International Conference on Multimedia Computing and Systems*, 7-11, June 1999, Florence, Italy. Vol 2, pp. 647-653.

[22] Ockerbloom, J. (1999). *Mediating Among Diverse Data Formats*. Ph.D. thesis, Carnegie Mellon University, January 1999.

[23] Salutation Consortium (1998). *White Paper: Salutation Architecture: Overview*, http://www.salutation.org/whitepaper/originalwp.pdf.

[24] Saund, E. (1999). “Bringing the Marks on a Whiteboard to Electronic Life.” Published in *Cooperative Buildings: Integrating Information, Organizations, and Architecture. Second International Workshop* (CoBuild’99). Pittsburgh, USA, October, 1999 (Springer Verlag Lecture Notes in Computer Science 1670).

[25] Scribner, K., Stiver, M.C (2000). *Understanding SOAP: The Authoritative Solution*, SAMS Press, ISGN 0672319225, January 15, 2000.

[26] Sun Microsystems (1997). *JavaBeans Specification*. Graham Hamilton, ed. http://java.sun.com/products/javabeans/docs/beans.101.pdf. July 24, 1997.

[27] Sun Microsystems (1999). *Jini Discovery and Join Specification*, January, 1999.

[28] Universal Description, Discovery, and Integration Consortium (2000). *UDDI Technical Whitepaper*, September 6, 2000. http://www.uddi.org/pubs/lru_UDDI_Technical_White_Paper.PDF.

[29] Venners, B. (2000). *Jini Service UI Draft Specification*. http://www.artima.com/jini/serviceui/DraftSpec.html. April 24, 2000.

[30] Waldo, J. (1999). “The Jini Architecture for Network-centric Computing,” *Communications of the ACM*, July 1999, pp. 76-82.

[31] Weiser, M., and Brown, J.S. (1996). “Designing Calm Technology.” *PowerGrid Journal* v. 1.01. http://www.powergrid.electriciti.com/1.01.

[32] Wollrath, A., Riggs, R., Waldo, J. (1996) “A Distributed Object Model for the Java System,” *USENIX Computing Systems*, vol 9, November/December, 1996.