Challenge: Recombinant Computing and the Speakeasy Approach

W. Keith Edwards, Mark W. Newman, Jana Sedivy, Trevor Smith
Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304
{kedwards, mnewman, sedivy, tfsmith}@parc.com

Shahram Izadi
University of Nottingham
Nottingham, UK
sxi@cs.nott.ac.uk

ABSTRACT

Interoperability among a group of devices, applications, and services is typically predicated on those entities having some degree of prior knowledge of each another. In general, 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, discovery of new services and devices, and contextual awareness.

Categories and Subject Descriptors

D.2.12 **[Software Engineering]**: Interoperability—*distributed objects*

General Terms

Design, Human Factors, Standardization

Keywords

Serendipitous interoperability, mobile code, recombinant computing, Speakeasy.

1. INTRODUCTION: PRIOR KNOWLEDGE AS THE BASIS FOR COMMUNICATION

The worlds of mobile, pervasive, and ubiquitous computing assume that we will be surrounded by a wealth of intelligent, interconnected devices and services. Many of the scenarios put forth by researchers and developers in these communities implicitly depend on the ability of these devices and services to be able to interconnect and interact with each other, easily and fluidly.

Such scenarios, however, beg a number of important architectural questions: how will these devices and services be able to interact with one another? Must we standardize on a common set of protocols that all parties must agree upon? Will only the devices and services that are explicitly written to use one another be able to interoperate, while others remain isolated?

Fundamentally, in order to interoperate, two entities must have some shared agreement on the form and function of their communication; this agreement defines the *interfaces* each entity exports, and which are known to the other party. A web browser and a web server, for example, must agree on set of common protocols (HTTP), as well as the format and semantics of any data exchanged over via those protocols (usually HTML or XML). Web services and their clients must agree upon known service interfaces, generally expressed in the Web Service Description Language (WSDL) and invoked via Simple Object Access Protocol (SOAP).

MOBICOM'02, September 23-26, 2002, Atlanta, Georgia, USA.
Copyright 2002 ACM 1-58113-486-X/02/0009…$5.00.

In all of these cases, there is *a priori* agreement on the form (syntax) and function (semantics) of communication, and this agreement is *built in* to each party to the communication. In other words, the respective developers of these programs have coded into them the ability to use a particular handful of protocols, and an "understanding" of the semantics of a handful of data types.

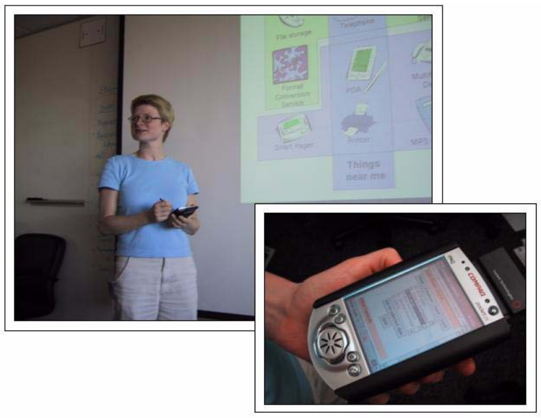


Figure 1 A user gives a presentation using Speakeasy. The network-enabled projector, and the filesystem on which the presentation resides, are Speakeasy components. The wireless PDA used to set up the connection automatically downloads presentation controls.

Clearly, if we hope for a future in which arbitrary devices and services can interoperate, there must be some prior agreement of the details of communication among these devices and services. The question that is at the core of this paper is: at what level should this agreement take place?

This paper describes an approach to interoperability, and an architecture called *Speakeasy* that embodies this approach, based on three premises: fixed domain-independent interfaces, mobile code, and user-in-the-loop interaction. We believe that this combination of premises, which we call *recombinant computing*, can support "serendipitous" interoperability—the ability for devices and services to use one another with only very restricted prior knowledge. The term "recombinant computing" is meant to evoke a sense that devices and services can be arbitrarily combined with, and used by, each other, without any prior planning or coding.

In the sections that follow we present our rationale for the premises upon which our approach is based, and why we believe that they are key to providing the ability for devices and services to offer serendipitous interoperability. After this, we present the Speakeasy system in more detail, including our experiences with the architecture and applications, based on two years of use, evaluation, and refinement. We conclude by examining the limitations of this approach, and the challenges that it raises.

2. RECOMBINANT COMPUTING: THREE PREMISES

In this section we discuss our rationale for the premises on which we have based our architecture.

2.1 A Small, Fixed, and Generic Set of Interfaces is the Key to Ubiquity

There are a number of commonsense approaches one might take to guaranteeing interoperability among a set of networked entities. One approach—common among current devices and many software services—is to agree upon one or more specific, *domain-dependent* interfaces. These are typically protocols that are particular to the domain of functionality of the device or service; for example, IPP for printing; CIFS or NFS for file storage; and many application specific protocols defined through various "interface definition languages" such as WSDL, RPCGEN, and IDL.

The problem with such interfaces, of course, is that there are so many to choose from: the range of such interfaces means that there are always likely to be specific interfaces that a particular entity cannot understand. My older PDA, for instance, has an infrared port, but cannot communicate with my IR-equipped phone, because the PDA does not understand IRCOMM, and there is no easy upgrade available to provide it with this new functionality. In other words, the required prior agreement on form and function of communication is not present in this case.

A second approach is the use of *domain-independent* interfaces. The web—which is inarguably the most successful example of interoperability among a wide variety of clients and services—uses this approach. The web achieves interoperability by relying on common knowledge of a handful of largely domain-independent protocols and data types. By "domain independent" we mean that all interactions between a client and a server are "funneled" into HTML over HTTP; there are no separate protocols for web cams versus personal home pages versus e-commerce pages.[1] Even though HTML and HTTP were conceived as being specifically for document delivery, they have been easily repurposed for delivery of information and control, independent of any particular application. In this sense, they represent a highly *generic* interface, capable of being applied to many uses. Generally, the promise of interoperability on the web is predicated on there being a *small* and *fixed* set of protocols and data types known to all entities, and which are *generic* enough to be easily adapted to new needs.

We believe that such a small, fixed set of generic interfaces—presumed to be known to all parties in an interaction—is essential to supporting ad hoc interoperability. If, instead, we dictate that all parties must have prior knowledge of a potentially open-ended set of interfaces, one for each type of thing we expect to encounter, we remain in the same situation as before: there is likely to always be some particular specific interface that is unknown to a party in a communication. This leads to the first premise on which we have based our architecture:

Premise 1: Agreement on a small, fixed set of universally-known interfaces, which can be applied to multiple purposes, is a more fruitful avenue for ad hoc interoperability than requiring that each party have prior domain-specific knowledge of every type of entity it may encounter.

The key here is that it is the fact that the set of universally-known interfaces are *fixed* that affords interoperability. Applications written to use these interfaces are guaranteed that they will be able to interact—at least in a limited way—with any entity that exposes its behavior using such interfaces. Certain devices and services may, of course, support other interfaces, which may be used by parties that understand them. But such interfaces outside of the core set are not assumed to be universally understood, and are not a mechanism for serendipitous interoperability.

2.2 Mobile Code Allows Dynamic Extensibility

There are obvious problems that come from requiring widespread agreement on a single fixed set of generic interfaces, the most obvious of which is their "one size fits all" nature. The particular interfaces in use—while perhaps sufficient for many needs—may not be optimal for all cases. And of course, fixing the set of interfaces limits the possibility of future evolution of the communication protocols between entities.

Mobile code can potentially provide the dynamism missing in a simple agreement on a fixed set of interfaces. Mobile code is the ability to move executable code across the network, on demand, to the place it is needed, where it is executed securely and reliably. The power of mobile code is in its ability to extend the behavior of entities on the network dynamically. In other words, part of the agreement that must occur between two interacting entities moves from *development time* to *run time*.

The most familiar systems that use mobile code today are Java's Remote Method Invocation facility [26], and Jini [24] (which itself uses RMI's semantics for mobile code), although mobile code is also possible using the bytecode language runtime which supports the languages in Microsoft's .NET platform, such as C# [15].

At runtime, these systems deliver objects whose implementations may not exist in the receiver. The type of the received object is assumed to be known to both the sender and receiver, although the particular implementation that is received is *specific to the sender*. As an example, a service may implement a "printer" type, which is known to its clients. The client, however, does not need to agree with—or even *know about*—the service's particular protocol. The knowledge of the particular protocol the service speaks is hidden within the mobile code transmitted to the client.

Mobile code provides a degree of indirection to the agreement that occurs between two communicating entities. Without mobile code, entities must agree on *all* specific wire protocols for communication. With mobile code, entities agree on a bootstrapping protocol to acquire new mobile code, and—since the receiver must know how to use any received code—the type signatures of that code. As others have noted, mobile code allows the interprocess interfaces to function much more like interfaces in a programming language [25].

This leads to the second premise on which we have designed our architecture:

Premise 2: Mobile code allows parties to agree on a small, fixed set of generic interfaces that allow them to acquire new behavior as appropriate, rather than requiring prior knowledge of specific wire protocols, data types, and so forth.

Mobile code changes the character of the generic interfaces discussed in our first premise. Rather than defining a fixed protocol, such as HTTP, these interfaces now define the ways in which an entity on the network can be extended by its peers. In essence, these interfaces become "meta interfaces" that define how extension happens, rather than how communication itself happens.

Of course, although mobile code allows entities to agree upon interfaces, rather than specific protocols, those interfaces must still be known in advance. Jini provides an interesting example. In the Jini world, there are domain-specific interfaces for entities such as

[1] Of course, this situation has begun to change recently, with the advent of specific XML-based data description languages; one of the primary purposes of these is to allow more machine intelligibility of web content. Each XML schema, however, defines a new data format, which may not be known to all parties on the network. Research into semantic understanding of such formats notwithstanding, we believe it is an open question how much interoperability will exist in such a web.

printers and file servers. Clients of such services can talk to any service, regardless of protocol, as long as it implements the known interface—but they must still agree on the interfaces. Systems such as Jini require that clients understand the type of each and every service they encounter, and—at least in current Jini services—each type of service is accessed via very particular, service-specific interfaces. These systems do not provide a set of fixed, generic interfaces for extensible behavior designed to accommodate serendipitous interoperability.

2.3 The User is Central in a Domain-Independent World

Although mobile code can provide greater flexibility to domain-independent interfaces than can a simple agreement on protocols, there is an additional implication inherent in the use of such interfaces: applications written against such domain-independent interfaces will not know the semantics of the entities with which they are interacting.

As an example, if my PDA is built to use a specific interface for printing, I can assume that the developer of my PDA understood *what it means* to print: what the semantics of printing are and when it is appropriate to do so, as well as printer-specific notions such as duplex, landscape, and so on. All of these notions are *implicit* in saying that my PDA is built to use a printer-specific interface. But this domain-specific knowledge is lost if we expect parties in a communication to expose only generic interfaces to themselves.

Our claim is that this dilemma is inherent in the approach, and goes back to the initial premise of using generic interfaces. We *cannot* expect a given device to be required to understand the semantics of each and every type of device with which it comes into contact—this would limit interoperability to only those sorts of entities explicitly known by the device.

If, then, we posit a world in which entities communicate using generic interfaces, and are not expected to have special knowledge of the semantics of any particular *type* of thing they may be able to use, this leads to our third and final premise:

Premise 3: Users will be the ultimate arbiters that decide when and whether an interaction among compatible entities occurs. This is because, in the absence of programs being hard-coded to understand particular entities, only human users can be relied upon to understand the semantics of these entities. This arbitration will largely happen at run time, rather than development time.

By acknowledging this premise, we claim even though a PDA might know nothing about printers specifically, it should still be able to use a printer if it encounters it. The PDA would never simply use an arbitrary entity it encountered, however, as it would not know whether the entity printed, stored, translated, or threw away data sent to it. Instead, the PDA would likely inform the user that a device calling itself a "printer" had been found, and leave it to the user to make the decision about when and whether to use it.

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 arbitrary interoperability; humans can, presumably, "understand" new entities 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.

2.4 Summary of our Premises

To summarize, our approach is based in a thread of argument that begins with the belief that a small, fixed set of interfaces, expected to be understood by all parties, is essential for interoperability. By their nature and because of the limited number of such interfaces, they will be independent of any particular application domain.

A naive approach of standardizing on a common set of protocols and data types, known to all parties, is potentially overly restrictive however, and mobile code is a solution to this problem. Therefore, these generic interfaces are recast as interfaces to enable dynamic extension, rather than interfaces for communication directly.

Further, if we presume a world in which entities on the network will have only limited knowledge of each other, we cannot expect them to be able to programmatically encode the semantics of all the many types of devices and services they may encounter. Therefore, it is implicit that users must be "in the loop"—able to make informed decisions about the entities they encounter, and provide the semantic interpretation that is otherwise missing, since semantics are no longer encoded in particular application-specific interfaces.

The next section discusses our exploration of an architecture based on these three premises.

3. SPEAKEASY: A FRAMEWORK FOR RECOMBINANT COMPUTING

Although the premises above argue for a small, fixed set of generic interfaces, they say nothing about *what* these interfaces should look like. We believe there are potentially a number of different "styles" of recombinant interfaces. For example, one might argue for a set of interfaces that mirror the classical computational functions of input, output, storage, and processing. An entity using such a set of interfaces might be extensible along each of these dimensions.

The particular collection of interfaces we have developed originated in our ideas about how best to satisfy the premises outlined earlier, and evolved based on scenarios, iterative design, and evaluation with actual users, over a period of two years. While we do not claim that the interfaces used by Speakeasy represents the only useful set of such interfaces, we have gained enough experience with them to believe that they afford great expressivity.

All Speakeasy devices or services—which we call *components*—implement one or more of a small number of interfaces that fall into four categories. These are the fixed, generic interfaces we expect all Speakeasy components and clients to understand:

- Data transfer: how do entities exchange information with one another?
- Collection: how are entities on the network "grouped" for purposes of discovery and aggregation?
- Metadata: how do entities on the network reveal and use descriptive contextual information about themselves?
- Control: how do entities allow users (and other entities) to effect change in them?

The data transfer and collection interfaces were motivated, respectively, by a desire to provide independence from particular communication and discovery protocols; they use mobile code to allow clients to evolve to support new such protocols. The metadata and control interfaces are a means of keeping users "in the loop." The metadata interface provides descriptive information about a component, while the control interface allows a user to control component aspects not represented through the other interfaces.

All of these interfaces share some stylistic concepts in common. Most importantly, each interface is used to return to its caller a "custom" object, specific to the component on which the interface was invoked. While each of these custom objects implements a type assumed to be known to the client and specific to its function (the data transfer interface returns a data transfer custom object, and so on), their implementations are specialized by the component that returns them, using mobile code that executes in the requesting application. Thus, clients invoke the data transfer interface on a component to acquire a custom object that extends their behavior to transfer data with the component. These custom objects represent the "inflection points" in the interfaces—the places in which mobile code is used to change the behavior of a client.

A second property common to all of these interfaces is that the custom objects returned to clients do not remain valid indefinitely. Instead, custom objects are *leased* to the clients that request them.

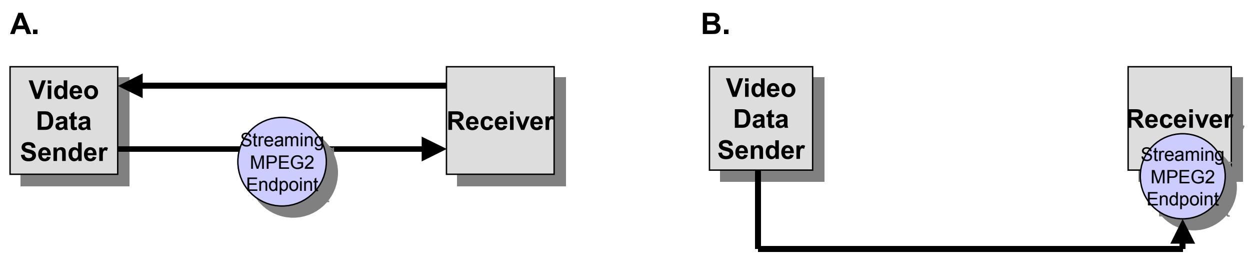


Figure 2 Data transfers in Speakeasy use source-provided endpoint code. In A, a receiver initiates a connection, which causes a source-specific endpoint to be returned. In B, this endpoint is used to fetch data from the source using a source-private protocol.

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 [7]. 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.

A final common property is that any client that uses the above interfaces to request a custom object must provide information about itself, in the form of a metadata custom object, described later in the section on contextual metadata. This mechanism allows components to know about who is requesting their services.

Again, these particular interfaces evolved during the course of the project through trial, error, and evaluation. Our research goals were to evaluate whether such a small, fixed set of interfaces would allow an application—once written against these interfaces—to interact with virtually arbitrary components that come along in the future that exposes its functionality in these terms. In other words, such a software system should be able to use new components that appear in its environment without recompilation or update and, further, be able to do so using new protocols and behaviors provided by the components themselves.

In the next sections, we will describe the particular interfaces and the styles of interactions they afford. Afterwards, we will offer our experiences with using the interoperable components and applications we have built with the Speakeasy framework.

3.1 Data Transfer

Our data transfer interface provides a simple, generic mechanism to allow two parties to exchange arbitrary data with each other. Such a data connection may represent a PDA sending data to a printer, a camera storing a snapshot in a filesystem, or a laptop 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 (an adaptive streaming protocol with variable compression, for instance, versus a reliable, lossless protocol). This example points out the infeasibility of deciding on a handful of data exchange protocols that all components agree upon. Each component has its own semantics with regard to sending data, and these semantics will be reflected in their protocols. Therefore, it is infeasible to build in support for all potential protocols "up front."

Instead, our data transfer interface uses 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 object," as described in the previous section, which provides a sender-specific 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. The endpoint acts essentially as a capability, allowing the party that holds it to initiate a transfer using a sender-specified protocol (it also allows a holder to invoke various "signalling" operations that indicate that the transfer has completed, aborted, or failed, and to request notifications about changes in the transfer's state).

Once the endpoint code has been received, the actual transfer of data is initiated. The endpoint in the receiver 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 filesystem will store it as a file. 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.

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 via standard MIME [2] type names. Programs and users can use these types to select components that are capable of exchanging data with one another

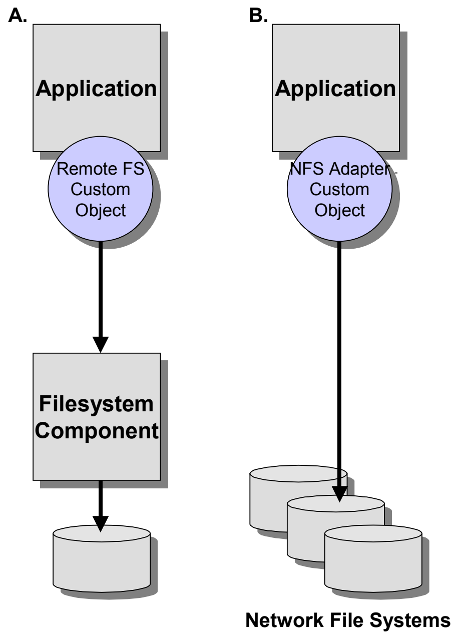
Figure 2 illustrates the operation. In the top part of the figure the receiver initiates a connection with a sender of video data, and the sender returns a custom endpoint to the caller (an implementation of a streaming protocol for MPEG2 data). This portion of the operation occurs using the "public" connection interfaces. After this, data is transferred to the receiver via the endpoint through a "private" protocol between the endpoint and the source.

The interfaces presented here allow any party to initiate the transfer. For example, a third party (such as a "component browser" application) can fetch an endpoint from a sender and provide it to a receiver, using the known data transfer operations on both. Such an arrangement would cause the receiver to read directly from the sender without involving the third party, using the sender's endpoint implementation.

3.2 Containment, Discovery, and Aggregation

The second generic interface in Speakeasy presents an abstraction that allows applications to use "collection-like" components. This *aggregation* interface provides the notion of groups of components logically contained within another. The core aggregate interface returns a custom object to the caller that provides access to the collection of components logically contained within the aggregate, as well as supporting facilities such as searching for components contained within the aggregate, and soliciting notifications about changes in the membership of the aggregate.

The design of this interface was motivated by both systems- and user-oriented desires. From the systems perspective, we wanted to allow Speakeasy applications to be dynamically extensible to new discovery protocols; we felt that this requirement was especially important in the mobile computing setting, in which a variety of discovery protocols may be used. These protocols would appear to the application to be "collections" of components found using the particular discovery protocol. From the user perspective, we


A.
Application
Remote FS Custom Object
Filesystem Component
B.
Application
NFS Adapter Custom Object
Network File Systems
C.
Application
Remote Discovery Custom Object
Discovery Protocol Bridge Component
D.
Application
Discovery Adapter Custom Object

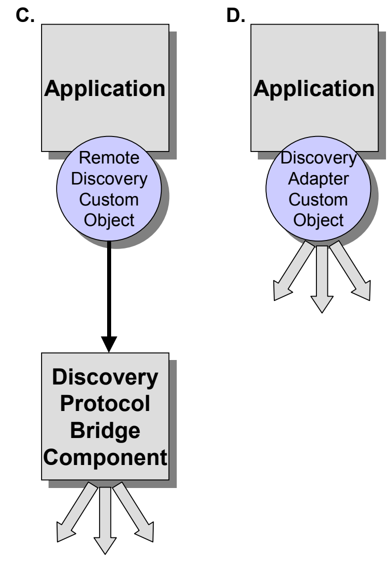


Figure 3 These four examples show how the single aggregate interface can be applied to multiple uses via component-specific custom objects. In each of these, a custom object implementing a known interface has been returned by a Speakeasy component to an application. In *A*, the custom object uses a private protocol to access a remote Speakeasy filesystem component, which is the aggregate that provided the custom object. In *B*, the custom object is an "adapter" that extends the application to speak the Network File System protocols—the protocol executes in the application "behind" the custom object's interface. In *C*, the custom object allows access to a discovery protocol "bridge" component. In this case, the remote component performs discovery and returns results to the application via the custom object. In *D*, the custom object implements a discovery protocol that executes completely in the application.

believed that it was important to allow components on the network to impose "groupings" that cluster related components together, rather than presenting all of them to the user at the same level.

There are a number of situations in which the interface is used. Filesystems, for example, are represented as *file aggregates* that organize components representing files. Such an arrangement allows an application written against the aggregation interface to "open" the filesystem to access the nested files and directories (which are themselves aggregates) contained within. Likewise, aggregates represent notions like discovery protocols and bridges to new networks, in which they provide access to components accessed using the protocols particular to them. In all of these cases, the aggregate will return (or generate) its list of contained components in a manner dependent on the aggregate itself.

Perhaps more importantly, since aggregates return custom objects to their clients, they have great flexibility in how they provide access to their logically-contained components. Each custom object is particular to the aggregate components that returns it, and has an aggregate-specific implementation that is downloaded from the aggregate and executes within the client.

This arrangement permits a number of interesting uses of this one simple interface. For example, the custom object can be a simple "shim" that communicates with the remote Speakeasy component that returned it via some private protocol. This can be used to create "discovery bridges," where applications access the results of a discovery protocol executing on a remote machine somewhere.

Alternatively, the aggregate custom object can add completely new protocols to the application, allowing it to communicate with legacy network services. For example, a component could return a Network File System "adapter" custom object, which executes the client NFS protocol directly in the application. This arrangement allows the client to access arbitrary NFS filesystems through the custom object. Likewise, custom objects that implement new discovery protocols can execute within a client. For example, a client can be extended to execute the Jini discovery protocols locally, by transferring to it a custom object that implements that protocol. Figure 3 shows an example of these various uses.

Finally, aggregates can act as bridges onto wholly new networks not natively "understood" by the application invoking the aggregate. They do so by "wrapping" legacy devices and services, to ensure that they are accessible. For example, a Bluetooth aggregate may use the discovery protocols within the Bluetooth stack [1] to find native Bluetooth devices and services, and create component "wrappers" for these that make them accessible in the Speakeasy world. In all of these cases, and no matter whether the aggregate custom object represents a shim to some backend service, or a complete protocol implementation, applications use the various custom objects through the single "container-like" interface implemented by them.

3.3 Contextual Metadata

Since our premises dictate that the semantic decisions about when and whether to use a component must ultimately lie with the user, we knew we must provide mechanisms to allow users to make sense of complex networked environments. For example, simply knowing that a component can be a sender or receiver of data provides very little utility if there is no other information that can be gleaned about the component. For this reason, one core Speakeasy interface returns a custom object that provides access to *contextual metadata* about a component. This custom object provides access to a set of attributes that might be salient for the component that returns it: its name, location, administrative domain, status, owner, version information, and so on.

Metadata custom objects have two uses in Speakeasy. The first is to simply describe the component that provided the custom object. A user, through an application such as a browser, would be able to organize the set of available components based on this metadata.

The second use is to provide components with information about their callers. All of the core Speakeasy interfaces require that their callers pass their *own* metadata custom object to the target component. For example, to invoke the data transfer interface, the initiator of a transfer must provide the sender with its metadata custom object. This mechanism allows components to know details about the entities that invoke their operations.

Our representations for contextual metadata are very simple: the well-known interface implemented by metadata custom objects allows access to a simple map of key-value pairs, with keys indicating the names of contextual attributes ("Name," "Location," and so on), and values that are themselves arbitrary objects, and may include digitally-signed data such as certificates. The set of keys is extensible, as we do not believe any fixed set is likely to support the needs of all applications or components. Likewise, we do not require nor expect that all applications will know the meaning of all keys, nor share a common metadata ontology. The goal of the metadata interface is primarily to allow sensemaking by users, and only secondarily to allow programs to use metadata in their interactions. Components and applications can agree on the syntax of interaction (the interface for retrieving contextual metadata) 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.

The use of component-provided custom objects in this interface leverages mobile code in much the same way that the data transfer interfaces can; this strategy supports a spectrum of underlying mechanisms for actually storing and transporting the metadata. The metadata custom object can implement component-specific protocols for accessing the actual metadata it provides. For example, it could communicate with some back-end process, providing up-to-date information at a cost in performance; alternatively, it could return "static" information without the need for communication with any other entity on the network.

3.4 Control

There are, of course, many aspects of component behavior that are orthogonal to simply sending and receiving data, or revealing new collections of components. While a printer, for example, may print data it receives, there are other aspects of a printer's behavior that aren't easily captured by the interfaces covered to this point. There is no notion of full duplex, or color versus black and white, or stapled output in the data transfer interfaces, for example. Nor should there be, since these are clearly concepts specific to printers.

Our approach is to expose such notions that aren't easily captured by the other interfaces to users, via component-specific custom objects that implement user interfaces. Applications agree on the mechanisms for acquiring and displaying such UIs, but have no knowledge of the particular controls provided by any UI. In the case of a printer component, an application could request the printer's UI custom object and display it to the user, allowing control over duplex, stapling, and so on. Applications need not have built-in support for explicitly controlling any specific component.

Applications that need to display the UI for a component can select from potentially any number of UI custom objects that can be provided by the component, by specifying the requirements of the desired UI. For example, a browser application running on a laptop might request a full-blown GUI, while a web-based browser might request HTML or XML based UIs. Applications interact with the custom objects via known interfaces that allow them to be displayed and used within the application.

This approach is similar in intent to the mechanisms used by Jini to associate user interfaces with services [23], although we do not require that the interface be stored in some lookup service known to the provider and consumer of the interface. The strategy 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 [9] or the UIML standard [8], 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.

Applications gain access to UI custom objects in two ways. First, they can request an "overall" UI interface for a component, using the core control interface in Speakeasy (one of the four interfaces listed earlier). This operation returns what is essentially an "administrative" interface for the component on which it is called.

The second way applications can acquire UI custom objects is asynchronously, as a result of some other operation performed on a component. We have mentioned that various of the custom object types in Speakeasy allow applications to register to receive asynchronous notifications from the component. While these notifications can return events that describe some simple state change, they can also encapsulate UI custom objects that the component "requests" an application to display. These UIs are *operation-specific*, as opposed to the "overall" UI that can be fetched directly from a component [17].

This use allows components to present UIs to users, who are likely sitting at a remote machine somewhere on the network, at the point that the component needs to interact with the user. For example, a video camera component could asynchronously send a control UI to an application at the point it is connected to a display. This UI may provide controls to pause the video stream, rewind, and so forth. Rather than controlling the overall behavior of the video source, this UI controls only aspects of that particular data transfer.

Like all custom objects in Speakeasy, UI custom objects are leased, and so do not remain valid indefinitely without continued proof of interest on the part of the application that holds them.

4. EXPERIENCES

Throughout the course of this research we have been motivated by a set of beliefs about how to maximize interoperability among a set of devices and services. Specifically, we believe that this type of dynamic, ad hoc interaction can be enabled by:

1. A small, fixed set of domain independent interfaces
2. The use of mobile code to add flexibility and dynamism to the interfaces
3. Allowing users to make the decisions about when and where to form connections between components.

The first premise raises the issue of whether or not it is realistically workable to design a small set of interfaces that is simple for developers to implement and at the same time affords rich enough functionality to be useful.

To explore this question, we have focused on deploying a fairly wide range of components while at the same time, focusing relentlessly on keeping the interfaces as small and simple as possible. We have built over two dozen so far, some fairly complex, as a means to vet and refine our interfaces. These include a whiteboard capture system; an internet radio system; a filesystem that can provide access to any Windows NT or Unix filesystem to other Speakeasy components; an instant messaging gateway; discovery components for Jini and Bluetooth; a Cooltown bridge that allows discovery and interaction with Cooltown devices that support the eSquirt protocol; printers; microphones; speakers; video cameras; a screen capture component that can redirect a computer's display to another component; and a number of others. We have also developed client applications including a "universal remote control"-style browser that allows access to components from any platform with a web browser, and a tool for collaboration [4].

The evolution of these components has often pointed to new directions in our architecture and consequently, our core interfaces have matured. We have not yet, however, had to move beyond the four basic interfaces described in this paper. Additional interfaces have been proposed (such as a "filter" interface that describes components that can act as both senders and receivers of data at the same time), but so far, we have been able to use the existing interfaces for all our implemented components and scenarios. In many cases, we believed that the addition of new interfaces (such as "filter") would come with an unacceptable increase in user confusion (such as having to move toward a more complex dataflow model of interconnection).

We pursued a similar strategy for exploring our second premise—that mobile code could be effectively exploited for providing the flexibility necessary to make the small set of interfaces workable. We have explicitly focused on creating components that require specialized transports (video, for example), use custom discovery protocols (IRDA and Bluetooth), or require knowledge of specialized datatypes (VNC data, for instance), in an effort to evaluate the efficacy of a mobile-code based strategy.

The system has been able to flexibly accommodate all of these, so far, by leveraging mobile code. Mobile code does, of course, bring with it an attendant set of challenges, specifically around security and runtime requirements (see the Challenges section, later in this paper, for more details). Our current focus has been on exploring the full use of mobile code as an enabling infrastructure, rather than on techniques for reducing runtime requirements or the security implications of a mobile code-based systems. Others in the research community are focusing on these challenges, and we hope to leverage their work where appropriate.

Our third premise, that users will necessarily be tasked with making the semantic interpretations of how to use the components around them, raised concerns about whether a system based on generic interfaces would quickly become unworkably complex. It was critical, therefore, for us to perform some kind of "sanity check" to assure ourselves that our basic model of interoperability was understandable to users.

To this end, we have undertaken a number of field studies of the system in use, as well as conducted evaluations of paper prototypes and mock-ups. With a handheld computer, test subjects were able to access Speakeasy components via a web application that allowed them to browse through all available components and form connections among them. Subjects were asked to perform a number of tasks, which required that users be able to grasp the basic concepts of discovery and containment, connection as a metaphor for data transfer, and other basic concepts provided by the Speakeasy infrastructure. Importantly, the handheld computer did not require any additional software to be installed on it other than a pocket web browser.

While a full discussion of our user studies is outside the scope of this paper (see [16]), we are encouraged by the results. Users seemed able to make sense of the highly generic world presented by Speakeasy, although there were a number of concrete directions that these studies made us aware of, including the need to be able to organize components based on their metadata in multiple ways. We believe these studies also present some interesting questions about interface metaphors for the mobile computing community at large. For example, do users think of PDAs as computers in their own right (that is, as "peers" of the devices around them), or as glorified remote controls (that is, devices "on the outside" of the things they are controlling)? Our findings suggest that both of these metaphors may be appropriate in different circumstances.

Our experiences with the architecture suggest to us that the recombinant approach can provide a powerful set of tools for interoperability, and further, that such a form of interoperability can be usefully understood and acted upon by users. In the near term, we plan to continue our current evaluation strategies by creating more components and applications for a range of domains, evaluating them under real use, and refining our core interfaces accordingly.

5. RELATED WORK

There are a number of systems that address one or more of the premises of recombination that we have laid out in this paper, but to date Speakeasy is the only one that addresses all three.

The Web Services standards (SOAP [19], UDDI [22], and WSDL [3]), as well as more traditional remote procedure call systems such as TI-RPC [20] and CORBA [18], represent agreements on sets of protocols and message formats, but fall short of specifying domain-independent interfaces, though such interfaces could certainly be employed on top of these frameworks. The languages in Microsoft's .NET [21] framework can potentially support mobile code, and as such might serve as a reasonable framework on which to implement a recombinant system. All of these systems assume that developers, rather than users, will make decisions about what sets of things to make interoperable and so do not satisfy our third premise.

Sun's Jini technology [24] makes extensive use of mobile code, both to insulate clients from the particulars of the services they communicate with, and to support service evolution. The current interfaces offered by Jini services, however, resemble the domain-specific styles seen in more traditional remote object systems—there are specific interfaces for each type of service, which clients must have prior knowledge of. As such, it does not address the issues of supplying a small set of domain-independent interfaces, nor does it focus on the end-user aspects of combining services.

Universal Plug and Play [14] does emphasize the ability of the user to decide when and what to connect together. However, UPnP supports interoperability through standardization on a fixed set of protocols and domain-specific interfaces, and does not leverage mobile code to allow extensibility to new protocols not already known by clients. The UPnP consortium is defining sets of fixed interfaces for different types of devices that are likely to appear in a networked home environment, such as audio-visual equipment, printers, and home appliances. Applications will be written to have knowledge of these domain-specific interfaces. For example, an application might know how to use a printer and a scanner but may well be unable to take advantage of a display, as well as be unable to take advantage of a new version of a scanner with (possibly proprietary) extensions to the basic scanner interface.

HP's Cooltown project [12] is concerned with extending web presence to devices and services situated in the physical world. As discussed earlier, the web is an excellent example of our first premise in that HTTP provides a small, fixed, universally known interface to a large number of services. By relying on web standards, Cooltown benefits from the ubiquity of the web, but is also bound by the web's limitations. These include reliance on the data types and protocols commonly used in the web. Further, the web suffers from the difficulty of supporting interoperation among programs using web protocols, which is basically the problem Web Services is trying to address. The web, and by extension Cooltown, focus on interactions between a client browser and a server; "third-party" interactions between two servers are not easily supported.

The iRoom [5] and Appliance Data Services [10] projects at Stanford provide ad hoc interoperability of loosely coupled services by allowing them to share data through tuplespaces [6]. In the case of the iRoom, the shared data are events, whereas in the case of Appliance Data Services, tuples are used to describe the routing of data and commands from appliances (devices) to services (which might be infrastructure services or other appliances). These systems depend on prior agreement on the format of the tuples, and so do not really address the issue of describing a small, fixed, domain-independent set of interfaces. This also means that users won't be involved in defining new combinations of services—only combinations among sets of services that have previously agreed upon tuple formats and semantics.

Ninja's Automatic Path Creation [13] and the closely-related Service Composition work at Stanford [11] take a dataflow approach to service composition, and come the closest of any of the work we have described to the model of recombination presented in this paper. In particular, Kiciman, Fox, et al.'s stated goal of “zero-code” composition [11], is strikingly similar to the goal of placing the user in the loop of deciding when and how to carry out interoperation. However, the data flow model and powerful primitives for service composition seem to afford significant complexity that many users may not wish to deal with. It may be possible that some of this complexity could be hidden by toolkits or libraries in order to make a system that could be reasonably managed by end users. These systems also take a different approach to dealing with potential protocol and data type mismatches, namely by introducing nodes in the service composition path that transcode from one protocol/type to another, rather than Speakeasy’s approach of using mobile code.

6. THE CHALLENGES OF RECOMBINANT COMPUTING

We believe that the approaches outlined here can provide an effective path to ad hoc interoperability among devices and services with only limited knowledge of each other. However, each of the premises upon which recombinant computing is based lead to a number of challenges.

First, the need for a fixed set of generic interfaces leads to the challenge of *what* exactly those interfaces should look like. The challenge here is to create a set of interfaces that are flexible and rich enough to be useful, and yet are still minimal enough that they can be adopted by application writers easily. Speakeasy provides an exploration of one particular set of interaction “patterns” for data transfer, discovery, and so on, but certainly there are others, which may be better or worse for different domains.

Second, the reliance of the recombinant computing approach on mobile code brings an attendant set of problems, which run the gamut from additional runtime requirements to security. We believe that security is the most important technical barrier to adoption of mobile code-based systems. The security challenges here lie in the areas of runtime security management that allows users to restrict the actions of downloaded code, flexible authentication, and access control.

The final challenge, and the one that we are the most fundamentally interested in, is the issue of usability. As recombinant computing necessarily puts the user in a place of primacy, we need to better understand how to achieve real usability in such a world. Usability goes beyond simple menu design and icon selection, of course. The fundamental architectural concepts of a system influence—and to some degree even determine—the styles of interaction that can be created on top of an infrastructure. How do the choices about interfaces, and the assumptions about semantics, influence the user experience of working in a recombinant world?

These challenges represent limitations inherent in the approach outlined here as recombinant computing. We believe that the benefits of our approach outweigh these limitations, but more work is needed to fully understand this point in the space of architectures.

7. REFERENCES

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

[2] Borenstein, N., and Freed, N. (1992). “MIME (Multipurpose Internet 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. Jan. 23, 2001.

[4] Edwards, W.K., Newman, M., Sedivy, J., Smith, T. (2002). “Using Speakeasy for Ad Hoc Peer-to-Peer Collaboration,” *Proceedings of CSCW 2002*, November 16-20, 2002, New Orleans, LA.

[5] 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.

[6] Gelernter, D. (1985) “Generative Communication in Linda.” *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80-112, January, 1985.

[7] 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.

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

[9] 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.

[10] Huang, A., Ling, B., Barton, J., and Fox, A. (2001). “Making Computers Disappear: Appliance Data Services.” *Proceedings of MobiCom 2001*, Rome, Italy. July, 2001.

[11] Kiciman, E., Melloul, L., and Fox, A. (2001) “Towards Zero-Code Service Composition.” Position paper for *Eighth Workshop on Hot Topics in Operating Systems* (HotOS VIII), Elmau, Germany, May 2001.

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

[13] Mao, Z.M., and Katz, R. “Achieving Service Portability Using Self-Adaptive Data Paths.” *IEEE Communications*, Jan. 2002.

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

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

[16] Newman, M., Sedivy, J., Neuwirth, C., Edwards, W.K., Hong, J., Izadi, S., Marcelo, K., Smith, T. (2002) “Designing for Serendipity: Supporting End-User Configurations of Ubiquitous Computing Environments,” *Proceedings of DIS 2002*, June 25-28, London, UK.

[17] Newman, M., Izadi, S., Edwards, W.K., Sedivy, J., Smith, T. (2002) “User Interfaces When and Where They are Needed: An Infrastructure for Recombinant Computing,” *Proceedings of UIST 2002*, October 27-30, 2002, Paris, France.

[18] Object Management Group (1995). “CORBA: The Common Object Request Broker Architecture,” July 1995, Rev. 2.0.

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

[20] Sun Microsystems (1997). *ONC+ Developers Guide*. August, 1997.

[21] Thai, T., Lam, H. (2001). *.NET Framework Essentials*. O’Reilly and Associates, June, 2001.

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

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

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

[25] Waldo, J. (2001). “The End of Protocols.” *Java Developer’s Connection.*

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