User Interfaces When and Where They are Needed: An Infrastructure for Recombinant Computing

Mark W. Newman, Shahram Izadi[*], W. Keith Edwards, Jana Z. Sedivy, Trevor F Smith

Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
{mnewman,kedwards,sedivy,tsmith}@parc.com

[*]Mixed Reality Lab
University of Nottingham
Nottingham NG8 1BB UK
sxi@cs.nott.ac.uk

ABSTRACT

Users in ubiquitous computing environments need to be able to make serendipitous use of resources that they did not anticipate and of which they have no prior knowledge. The Speakeasy recombinant computing framework is designed to support such ad hoc use of resources on a network. In addition to other facilities, the framework provides an infrastructure through which device and service user interfaces can be made available to users on multiple platforms. The framework enables UIs to be provided for connections involving multiple entities, allows these UIs to be delivered asynchronously, and allows them to be injected by any party participating in a connection.

KEYWORDS

Speakeasy, recombinant computing, ubiquitous computing, asynchronous user interfaces

INTRODUCTION

Imagine the following scenario: a corporate researcher is visiting a university to give a talk. She plans on "traveling light," with only a minimum of hardware—bringing only a PDA and leaving her heavy laptop at home. Upon arriving at the university, our researcher needs to use a number of locally available resources—including a projector and a large-screen display—as well as resources back on her "home" network—notably, her home directory on the file servers back at her institution, where her slides reside.

The researcher will have to plan a number of steps in advance in order to give her presentation in this manner, including ensuring that her slides are accessible from the site of the talk (by emailing them to a colleague, placing them on her PDA, or putting them on an external server, perhaps), making sure that the correct software versions are in place at the site of the talk, and so on. Once she arrives, the unforeseen requirements of her audience may require her to adapt to use new local resources. For example, doing a video telecast of her talk for her colleagues at home, or providing printouts for local attendees, may require finding additional resources, installing drivers, running cables, and so forth. Overall, the process is likely to be one fraught with the frustrations of software incompatibility, communication problems, version mismatches, and driver installations.

This scenario illustrates the problems with being able to take advantage of ad hoc resources in our environments: we are constrained by the lack of fluid interconnection between devices and services, and by the need for advance planning in order to overcome obstacles of interoperability.

We believe the solution to these problems lies in an approach called *recombinant computing* [3], which enables devices and services on a network to be fluidly "recombined" with no advance planning, even when those devices and services have only very limited advance knowledge of one another. We have built the *Speakeasy* infrastructure for recombinant computing, which allows end users to easily assemble new combinations of functionality, based on the resources they find at hand.

Speakeasy addresses several challenges inherent in allowing devices and services to interoperate with a minimum of a priori knowledge of each other, such as how services discover one another and how they transfer data among themselves. In this paper, we focus specifically on the UI challenges of such environments, and present the UI architecture in Speakeasy that addresses them. Speakeasy's UI architecture provides an infrastructure in which users can control arbitrary devices and services on a network, through custom user interfaces provided by the devices and services themselves. The mechanisms used by Speakeasy allow for arbitrary UI code to be delivered, either synchronously or asynchronously, to a client device. Further, each party involved in a combination can detect failures and partitions of the other parties—a requirement in a distributed networked setting.

The roadmap of this paper is as follows. In the next section, we present a high-level overview of the Speakeasy framework as a means of providing the background necessary to understand the UI architecture. After this, we investigate the requirements for a UI infrastructure designed to facilitate user control over highly dynamic networked environments. Next, we discuss the Speakeasy UI architecture. The major contribution of this paper is in

the design of a protocol for the asynchronous delivery of user interfaces across a network to an application, without requiring that the application have any special advance knowledge of the UI it receives, and in such a way that connectivity failures are detected and dealt with properly. We conclude with a discussion of related work, and some future research directions.

BACKGROUND: THE SPEAKEASY RECOMBINANT COMPUTING FRAMEWORK

The main goal behind the Speakeasy framework is to enable a vision that we call *recombinant computing* [3], which dictates that devices and services with little or no *a priori* knowledge of one another should be able to interoperate. The recombinant computing vision is based on three core principles: (1) employing a small, fixed set of generic interfaces, (2) using mobile code to allow components to extend one another's behavior at runtime, and (3) keeping the user in the loop in deciding when and how components should interact with each other.

Speakeasy's approach to accomplishing the recombinant computing vision is to cast all devices and services on the network as "components" that implement a small, fixed set of programmatic interfaces which allow them to be "snapped together" at runtime by users. These interfaces describe how clients discover available components, how data is transferred between components, and (as we discuss in detail in this paper) how user interfaces are delivered to clients. These simple, narrow interfaces are made much more powerful by the fact that components can deliver mobile code (portable code that can be downloaded over a network to a client, where it is executed) to extend the behavior of applications that use them, in order to adapt the applications to new functionality.

The principle of allowing *users* to determine when and where interactions should take place [11] is a key part of what enables the set of interfaces to remain small and generic. This is accomplished by focusing the interfaces on the syntax of inter-component communication while leaving the determination of semantics to the user. To take the trivial example of printing a document, a printer component might appear to other components and to applications as simply a component that can receive certain types of data (e.g. Postscript). These clients would not need to know anything about *printers*, only about generic *data sinks*. A user, on the other hand, would know the *semantics* of a "printer"—that it is a type of thing that prints data when it receives it, and be able to make a decision about whether or not to use it. Meanwhile the application, and the components involved, need know nothing more than the "syntax" of the interaction.

Connections and Data Transfer in Speakeasy

Before embarking on the detailed description of Speakeasy's UI framework, which is the focus of this paper, we will pause to give some background on Speakeasy's connection framework, which will be essential for understanding several aspects of the UI framework. Speakeasy regards a *connection* as an association among two or more components for the purpose of transferring data. Such a transfer may represent a PDA sending data to a printer, or a laptop computer sending its display to a networked video projector. A connection typically involves a component that is the sender of data (called the source), a component that is the receiver of data (called the sink), and an application that initiates the connection.

The Speakeasy connection framework leverages mobile code to allow components to be extensible in both the data communication protocols and the data types they can use. The first of these is accomplished through a mechanism called *session objects*; the second through a mechanism called *typehandlers*. Together, these two mechanisms allow, for example, a projector component to be able to receive and display streaming MPEG video data from a camera, without having to be expressly written to understand the streaming protocol, and without having to be expressly written to understand the MPEG data format.

These two mechanisms are described below, and form the basis on top of which much of our user interface framework is built. The remainder of this section briefly describes these mechanisms, as a basic understanding of them is necessary for motivating and describing our UI framework.

When an application wishes to connect two components, it first acquires a *session object* from the source component. The session object is a complete, serialized object—expressed using the Java language, in our implementation—that implements a known interface but whose concrete type is provided by the source component itself. In other words, each source can provide a specialized session implementation; the code for such specialized implementations will by dynamically downloaded on demand from the source by the holder of the session object.

Once an application has acquired a session object from the source, it can provide a copy of this same session object to the sink, through serialization and transmission over the network. This act initiates the connection between the two components. The sink delegates the fetching of data to the session object: the session object—since it was provided by the source—can use whatever data transfer protocol the source expects or prefers to use. Thus, the session object effectively extends the behavior of the sink to enable it to transfer data using a protocol dictated by the source. Once the connection is started, the application that initiated it is not directly involved in the connection, although the session object acts as a capability through which the application can abort a connection or monitor its status.

For example, in the scenario posed earlier, a video camera may transmit its data using a streaming protocol, which adapts to changing transmission rates by varying the level of compression. The projector need know nothing of this protocol. By acquiring a session object from the video camera, it acquires the ability to use this protocol.

Of course, an extensible mechanism for specifying *how* the data is obtained is useless if the receiver doesn't understand *what* the data is. Speakeasy's connection framework provides extensibility of data type handling, to allow components to perform certain operations on data types they may not "natively" understand.

Each Speakeasy component expresses the types of data it understands as a list of MIME types [1]. Additionally, components can express that they have knowledge of certain *programmatic* types. These are interfaces (in the programming language sense of the word) that the component understands. Components can provide concrete implementations of these interfaces that "wrap" the underlying raw data with an object that provides a known interface to using that data. These concrete implementations are called *typehandlers*.

For example, a projector component[1] may be explicitly written to understand certain types of data—perhaps JPEG and GIF images—and be able to display data in such formats. It might not, however, be written to understand other data types such as MPEG or PowerPoint. To allow for the ability to view these other types of data, the projector could be written to use objects that implement an interface known to it; this interface would be used by the projector to obtain a visual representation of data in an otherwise unknown format. In this case, the projector would advertise its compatibility as a list containing the standard MIME types `image/jpeg` and `image/gif`, and *also* a programmatic type named by its interface, such as `com.parc.speakeasy.Viewer`, for example.

A video camera source, then, could be written to provide not only its native MPEG data, but also a specialized typehandler implementation of the `com.parc.speakeasy.Viewer` interface. The typehandler will be transparently downloaded by the projector, where it retrieves the underlying data from the source in its raw MPEG format and then displays it.

Together, these mechanisms provide Speakeasy applications with two dimensions of flexibility: the ability to acquire new data transport protocols at runtime, and the ability to acquire new type handling behavior at runtime.

Figure 1 illustrates the three main phases of data transfer (labeled 1, 2 and 3 respectively). In the first an application acquires a session object from the source component. In the second, the application keeps a copy of the session object and passes another copy across the network to the endpoint of the transfer—the sink—which initiates a connection directly from source to sink (which the application can control). In the third, the source returns a typehandler to the sink (in this case, a typehandler capable of viewing MPEG data). This portion of the operation occurs using the "public" Speakeasy interfaces. After this, data is transferred to the sink through a "private" protocol between the session and the source.

[1] We imagine a "projector" component to be a service running on the network that provides accepts data from other components and displays it on a projector. In our current implementation, this means that a server is connected to the projector's VGA and serial ports and handles the display and control on behalf of the projector. In the future, we anticipate the existence of devices that would incorporate processing, networking, and projection into a single package.

UI REQUIREMENTS FOR RECOMBINANT COMPUTING

During the initial development of Speakeasy, it became clear that our model for ad hoc component use would require a new style of UI infrastructure. While a number of existing UI infrastructures focus on the needs of ubiquitous computing applications, we felt that these were not easily applicable to our model for a number of reasons.

In this section, we discuss the set of requirements that emerged from our initial investigations, and that drove our development of Speakeasy's UI framework.

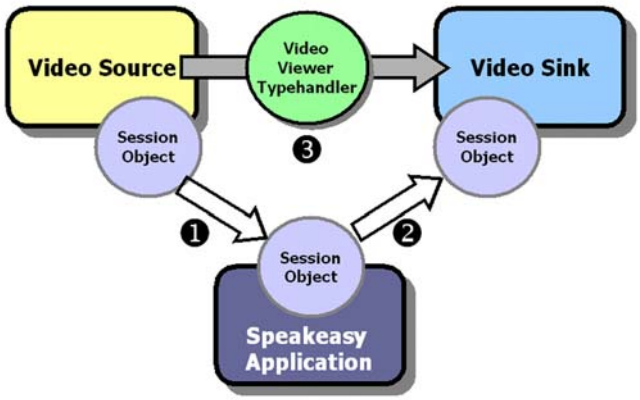

Figure 1: An example connection in Speakeasy. The application requests a session object from the source (1) and passes it to the sink (2). Then the sink uses the session to ask the source for a typehandler to carry out the transfer (3).

Requirement #1: Component UIs must be able to find their way across the network to the user that is using them, even when that user is only indirectly involved in the operation.

In the Speakeasy model, users largely effect change in the environment through the connection mechanisms outlined in the previous section. We envisioned that a user, through a "browser"-style application, would initiate connections between components. The application itself would not be directly involved as an endpoint in the connection, and would likely be running remotely from the components involved in the connection. So we required mechanisms for delivering interfaces from the external components involved in the operation to the remote client that initiated it. This is in contrast to the web, where the browser is an endpoint in the connection with the server. In the Speakeasy model, the application that initiates a connection may not itself be an endpoint, but yet must still have the ability to control the connection.

Requirement #2: Applications can be written to have only general knowledge of component UIs, and cannot be expected to have specialized knowledge of every type of component they may encounter.

A key premise of recombinant computing is to move away from building in *a priori* support for all conceivable devices and services into applications. Instead, we wanted applications to be able to obtain and use UIs for components they had not been specifically written to know about. Without this, the opportunistic styles of interaction envisaged by the project would be difficult to facilitate.

The ideas behind distributed UI approaches such as those found in the Jini [18] ServiceUI framework [16] lend themselves well to this requirement. The ServiceUI framework allows Jini services to provide user interfaces for a variety of clients across the network. These UIs can be downloaded on demand, whereupon they are presented to their users. Dependent upon deployment requirements, the underlying logic associated with the UI can either be downloaded alongside the UI and run locally or accessed remotely using RPC mechanisms such as RMI [19].

Requirement #3: Components must be able to asynchronously "push" UIs to clients at arbitrary points in a connection's lifecycle, while still allowing applications to "pull" UIs at certain times.

There are problems with approaches such as ServiceUI, however, most importantly that they present only a *pull* model, whereby UIs are explicitly requested by applications. It was clear to us that many components would support entirely *different* UIs that would need to be presented to the user at different points in the component's lifecycle. A printer on a desktop operating system, for instance, provides a "top level" UI that allows the user (or an administrator) to configure various defaults, queuing options, and so forth. But this same printer will typically provide a different UI that is presented when the user actually prints something; this UI allows the user to control number of copies, etc., for the *current act of printing*.

This dichotomy—of a top level *per-component* UI, versus an operation-specific *per-connection* UI—was present in many of the scenarios we envisioned, and thus our framework needed to support both. Frameworks such as ServiceUI are inappropriate for the rich interactions envisioned by Speakeasy, since they assume that applications know to request the right UI at the right time.

In particular, applications—since they are unaware of the semantics of any particular component—will not know *when* that component needs to have a per-connection UI displayed to the user. A secure filesystem, for example, may need to present a login panel to the user *before* any data is stored in the filesystem. A slideshow component may need to display controls for slide navigation *throughout* the duration of the slideshow. We could also easily imagine situations where the semantics of a particular component could dictate that an operation-specific UI be presented only under some exceptional condition. For example, a streaming video component might only present a UI to its user if the channel were experiencing unusual losses, perhaps allowing the user to throttle back the frame rate, or increase compression.

Requirement #4: Applications and components must be able to detect and recover from partial failure situations.

Since these UIs are displayed on machines remote from the entities they are controlling, applications and components may be exposed to *partial failure* situations. Such failures must be considered in any networked environment, and particularly in those where resources may be abruptly disconnected without first being able to inform their peers. Under these circumstances, a machine hosting the UI logic could find it difficult to determine if a remote peer has actually gone down or whether it is just slow in responding due to network latencies or if it is only momentarily disconnected. The result is that a large number of "stale" connections can be accumulated, holding valuable resources. Because of these problems, we imposed a requirement on our infrastructure that any party involved in a connection be able to detect failures in any other party, without the need for explicit disconnect or ongoing communication to signal a disconnection.

Requirement #5: The infrastructure must be able to support "aggregate" UIs, where different pieces of the whole are provided by different parties in a connection.

Finally, and as we will illustrate in more detail in our section on per-connection UIs, *any* party in a connection may have the need to "inject" a per-connection UI into the network. Therefore, the framework must support the ability to create "aggregate" user interfaces, potentially originating from multiple parties in the connection.

RELATED APPROACHES

A number of architectures satisfy some of the requirements listed above, but to our knowledge none exist that satisfy them all.

For example, there exist a set of architectures that allow applications to obtain UIs for entities about which they have no advance knowledge. The Web is perhaps the most successful example, and it succeeds by providing a simple, universal standard for data transport (HTTP) and description language for providing user interfaces (HTML). Effectively, if a user can discover the URL for interacting with a service, she can obtain its UI. A number of architectures [8, 9, 17] make use of web technology for providing a service's UI to users and applications that have no prior knowledge of it. Another set of architectures [4, 6, 7, 12, 13] extend this concept by providing abstract representations of interfaces that can be used to generate platform- and mode-specific user interfaces at runtime.

These architectures gain many of the advantages of the web, but also suffer from some of its weaknesses, such as the inability of users to easily connect two components when they themselves (or their devices) are not directly involved in the interaction. In other words, most web-based approaches implicitly assume that the browser itself, or the

device on which it is running, is one of the endpoints in a connection. The Speakeasy UI architecture additionally addresses the case where the device used to initiate and control a connection is separate from the entities being connected. In addition, these architectures are largely bound to the "pull" model of the web, and so are not suited to "pushing" interfaces towards users when user intervention is required during an interaction.

A notable example of an architecture that borrows heavily from the web, but extends its interactive capabilities, is Olsen, et al.'s XWeb [13, 14]. In addition to leveraging web-like capabilities to deliver UIs to clients with limited a priori knowledge of the services they control, it also allows interface aggregation and a limited form of asynchrony. XWeb's asynchronous support is limited to notifications of changes in a service's data; clients must be programmed to know how to deal with these changes. XWeb also has the notion that multiple parties (services and clients) might be involved in an interactive session—though it is not clear whether clients can easily direct the transfer of data among heterogeneous services, or whether the interactions are restricted to "control-like" interactions with services. It is also not clear what mechanisms are built into XWeb for dealing with partial failures.

Other systems, such as the ICrafter [15] framework for building interactive workspaces, allow users to obtain UIs that apply to a specific association among a set of components. In addition to supporting the generation of platform-specific UIs, ICrafter provides a notion of "aggregation" that allows UIs to be written for "patterns" of interfaces. In other words, a UI can be created that glues together the functionality of multiple components. However, this is somewhat different from our goal of allowing components to provide custom UIs for specific connections. In addition, the requirement to support asynchronous delivery of user interfaces to multiple parties as the needs of a particular connection change is not addressed by ICrafter.

As noted previously, the ServiceUI framework allows services to provide mobile code-based user interfaces for arbitrary "roles" (e.g. "main UI," "administrative UI," etc.) and modes (e.g., speech, GUI). While the use of mobile code allows virtually arbitrary user interfaces to be created (not just web forms, for example), these user interfaces are still limited to the "pull" model, provide no facilities for partial failure detection, and have no support for aggregate interfaces in the style of ICrafter.

USER INTERFACES IN SPEAKEASY

We developed the Speakeasy user interface infrastructure based around these requirements. This infrastructure has two main pieces to support both per-component and per-connection UIs. In this section, we present aspects that are common to both pieces, and briefly introduce each. The bulk of the remainder of the paper details the per-connection UI infrastructure, which we believe is the main contribution of this work.

In much the same way that the Speakeasy exploits mobile code for runtime protocol and datatype extensibility, the system also used mobile code to allow user interfaces to be downloaded on demand by applications where upon they can be presented to users. Applications need not have built-in support for explicitly controlling any component. Instead, this functionality is gained at runtime by downloading the mobile code for user interfaces associated with the desired services.

Since applications may be built using different modalities (e.g. GUI, speech, etc.), they 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 might request a UI that uses a full-blown GUI toolkit already present on the machine, while a web-based browser might request HTML-based UIs. Much like ServiceUI, this approach allows multiple UIs, perhaps specialized for different classes of devices, to be associated with a given component.

A drawback with this approach 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 in [4, 6, 7, 12, 13], and 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.

UIs for Components

All components in Speakeasy can provide one or more "per-component" user interfaces. Typically, these are "administrative" or "configuration" interfaces that govern the global behavior of the component. For example, a component representing an LCD display panel might provide a user interface to allow users to configure universal settings such as the display resolution, color depth, and so on.

These interfaces are always "pulled" by the application, at the demand of the user. Per-component UIs are the simplest form of UI supported by Speakeasy, in terms of demands on the infrastructure, and are similar to ServiceUI, although we do not require that the interface be stored on some lookup service known to both the provider and consumer of the interface, as is the case with ServiceUI.

UIs for Connections

Additionally, applications can acquire UIs as a result of connections between components. Unlike per-component UIs, these interfaces are pushed asynchronously to the client, wherever the client happens to be, even if the client machine isn't itself one of the endpoints in a connection.

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 camera, this UI controls only aspects of that particular data transfer.

In a recombinant world, where devices and services are often interconnected rather than used in isolation, we have found that these operation-specific UIs, termed *Controllers*, play a particularly important role. Most importantly, they allow users to shape the interaction during connections between components, and provide the necessary feedback regarding the state of the connection.

THE CONTROLLER UI INFRASTRUCTURE

At a high level, the controller infrastructure can be seen as a way in which a loosely-coupled set of applications and components can share user interfaces, without requiring that applications that receive those UIs have either advance knowledge about the types of UIs that will be received, or where those UIs may come from. This section describes the mechanics of the Speakeasy controller infrastructure.

Consider two or more Speakeasy components on a network that are interconnected for the purposes of data exchange. Typically, these components will be on different machines, and the connection between them will be initiated by a user working with a "browser" application on a third machine.

In our implementation, the browser initiates the connection by requesting a session object from the component that will be the sender of data, and then transmitting this session object to the component that will be the receiver, as described earlier. Thus, all parties involved have copies of the session object. The session object is used to knit together the set of applications and components that either are directly involved in the connection (i.e., as an endpoint), or are "interested" in the connection (i.e., can control the connection). The session object acts essentially as a "capability" [2] for controlling or receiving updates about the connection. All parties that hold the session are eligible to perform certain operations on the connection (such as stopping it), and can solicit notifications about changes in the state of the connection.

For the purposes of our discussion on the controller framework, any party that holds the session also has the capability to *add* a controller UI to the session. The act of adding a controller means that the party wishes UI to be displayed to the user. The party that adds a controller to a session becomes the *master* for that controller. This means that, in addition to receiving events about the state of the connection, it will also be eligible to receive events about the state of the controller it has added—whether a browser has displayed it, whether the controller has crashed, whether the user has dismissed the, whether the controller has finished naturally, or whether the computer displaying the controller has crashed or lost contact with the network. This step is illustrated in Figure 2.

When a controller is added, a notification is sent to all other holders of the session. This notification is in the form of a *controller event* that contains the serialized user interface, as well as details about the user interface platform on which it runs, and so on. An interested party, such as a browser, can then *take* the controller, which again notifies all interested parties about the change in state. By "taking" a controller, a browser is agreeing to display the controller to the user. A browser that takes a controller from a session is called the controller's *host*. Code in the host will notify the controller's master if the controller fails, is dismissed, etc. This step is illustrated in Figure 3.

The details of how events are propagated are transparent to components and applications themselves—all copies of the session object register to receive updates about the state of the session upon deserialization; this mechanism provides weak consistency among all copies of the session. Events are tagged with sequence numbers to facilitate duplicate and missed event detection.

More importantly for the purposes of the controller infrastructure, this mechanism allows any program that holds the session—even if it is not itself one of the components involved in a connection—to add or receive a user interface, no matter where it is on the network. The mechanism described here addresses our first requirement, that UIs be deliverable to "third party" applications that are not themselves endpoints in a communication.

Further, controllers can be added at any point during the lifetime of the session, not just at the start of the session, or before data transfer begins. Imagine a video camera that is connected to a viewer. If network congestion occurs, the camera may need to present controls to the user during the course of data transfer to allow the user to change buffering characteristics and so on. The protocol outlined here allows fully asynchronous delivery of UIs to support this style of interaction, addressing our third requirement, that UIs be deliverable at the time they are needed.

An application that "takes" a controller can specify the general contract that it requires of the controller—what the platform requirements of the controller are, and what interfaces the serialized user interface will implement. As long as there is agreement on these *general* details, a browser can take and use a controller without having to know the specifics of it. For example, a browser can simply know that a received controller creates a window using a particular GUI toolkit, without having any more particular knowledge of it. This approach addresses our second requirement, that applications have only generic knowledge of the UIs they may receive.

Further, any taken controller is *leased* by the host from its master. Leasing is a technique whereby the host must demonstrate continued proof of interest to continue using a resource (in this case, a controller) [5]. If a browser does not do this—because it has crashed, or became disconnected from the network, or does not follow the proper protocol for hosting a controller—the lease on the

controller will expire, and the master will terminate the controller. This mechanism allows all involved parties to know if a controller's host has lost contact with the network for some reason, so that they can clean up after it. Further, it allows these parties to mutually know this fact without further communication with one another. For example, if a user is controlling a presentation on a projector from a laptop, then simply turns off the laptop without first disconnecting from the projector, the system will detect this, terminate the connection, and allow the projector to be used once again. The use of such "soft state" satisfies our fourth requirement, that all parties in a connection be able to determine if a failure has occurred, without the need for any further communication between components.

Since sessions act as capabilities, any party that holds the session can add a controller to it. When multiple parties add controllers to a session, the browser assembles these into an "aggregate" UI, as described previously in our fifth requirement, which presents all UIs organized into a single window. In our current implementation, library code automatically handles the aggregation and presentation of controllers, as they are received by an application. Figure 5 shows one example of this, with two controllers on separate tabbed panes; other organizations are of course possible.

Design Discussion

In practice, all of the steps of the protocol are codified in libraries that reside in components, and in browsers, and take care of the details of implementing the controller infrastructure. The end result is that any party in a session can request that a user interface be displayed to a user sitting at a browser anywhere on the network. This user interface will be "pushed" to the browser at runtime. Further, when the browser actually displays the user interface, all interested parties can be aware of this fact, and can follow the state of the controller (whether it's still active, and so on), and perhaps take action accordingly.

The notification and leasing mechanisms together provide components with information about whether their user interfaces have failed to be displayed to a user, and allows components to adapt their behavior accordingly. For example, a component may *require* that its user interface be displayed to the user, if the component is to be used; if no browser takes the controller, or if the user dismisses the controller, the component can detect this and drop the user's connection. This protocol also supports multiple simultaneous controllers being displayed or used by different browsers, and multiple simultaneous controllers being added by different parties involved in the session.

The requirement that any party be able to add controllers has proved crucial to Speakeasy. In a setting in which applications do not have specialized knowledge about the parties that may be involved in a session, we need mechanisms in which a range of parties can add user interface elements without increasing application programming overhead. In particular, the Speakeasy data transfer model, with its multiple components, downloaded typehandlers, and so forth, requires that any of these entities be able to add user interfaces when appropriate.

A source component may add a controller to allow a user to parameterize data transmission; for example, a camera may allows the user to alter frame rate, compression, and so forth. Likewise, a sink component may add controllers to a session; for example, a printer may display a dialog box giving control over various print options. Also, mobile code elements such as typehandlers may add controllers; for example, we have built a generic "viewer" component that can view arbitrary media types for which a typehandler is available. If a sender transmits a PowerPoint file to the viewer, the PowerPoint typehandler adds a controller with buttons to navigate through the slides. If a sender transmits a video file, the video typehandler adds a controller to allow pausing, stopping, etc., the video. This style of aggregate UI, built dynamically using the entities involved in a data transfer, is used ubiquitously in Speakeasy.

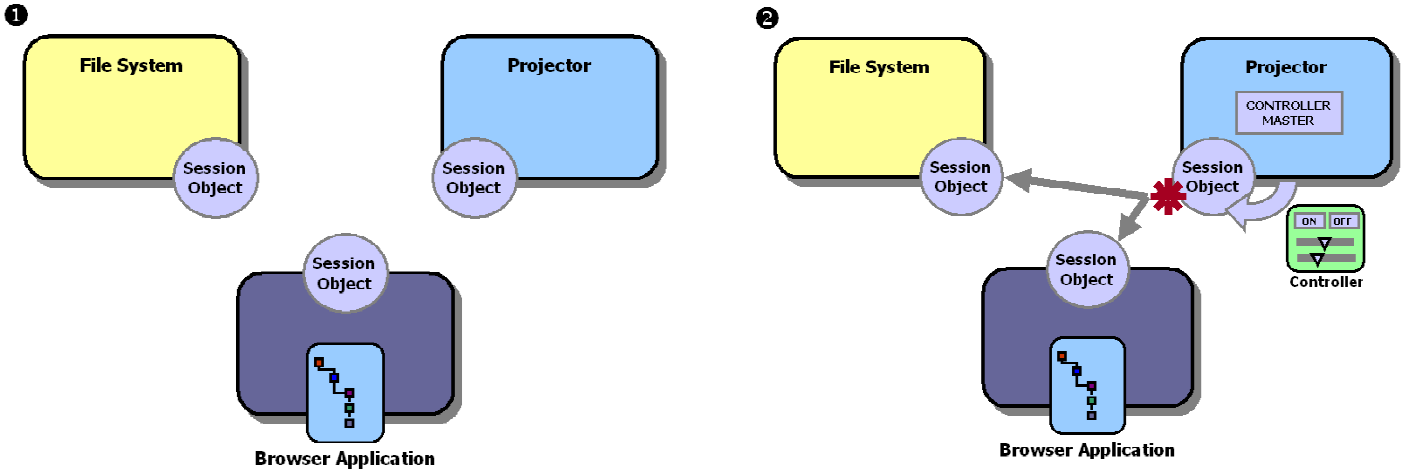

Figure 2: Two initial steps in the controller protocol. In 1, all parties gain references to a "session" object, which provides a capability to control a connection. In 2, one party (in this case the projector), "adds" a controller to the session. This causes "controller added" events to be sent to all other parties that have expressed interest in such events. The projector becomes the "master" for that particular controller.

EXPERIENCES WITH THE CONTROLLER FRAMEWORK

The architecture described in this paper has been fully implemented, along with a wide range of components and controllers that allow us to realize not only the scenario posed at the first of the paper, but also a number of other interesting uses.

Realizing The "Travel Light" Scenario Using Speakeasy

The scenario described at the first of this paper has been implemented through the use of a Speakeasy-aware browser application that runs on a PDA. Upon starting, the browser begins a discovery process to find all available components (as described in [3], Speakeasy is able to take advantage of a variety of discovery protocols, for example Jini or Bluetooth). This process reveals a number of components, including those that represent the local projector and the user's remote file space. The user can "open" the remote file space to locate the desired file. Once located, this file can be connected to the component representing the projector.

During the connection, the projector requests the typehandler for the `application/powerpoint` type from the source, since it does not understand PowerPoint data natively. This causes the typehandler code to be transmitted to the projector, where it is executed. The typehandler then retrieves the data from the source file in its "raw" format as PowerPoint data, and makes a local copy of it. It then opens and renders the file.

Upon establishing the connection, the remote filespace adds a controller to the session. This first controller is an example of a "passive" controller that simply presents the user with the progress of the data transfer using a progress bar UI widget. It does not directly provide the user with any control over the transfer operation–although the user can still terminate the transfer session by dismissing the controller. The controller is only active while the file is being copied over the network, after which time it is removed from the session, resulting in the UI disappearing from the user's PDA.

After the file transfer completes, the projector component adds a controller to the session. Again, as this controller is added, the browser (and all other interested parties) will receive an asynchronous event that encapsulates the controller's UI object. This UI provides controls for setting up the projector—powering it up, setting video mode, brightness etc (see Figure 5)—and is rendered on the PDA's display. The UI communicates over the network with the projector component. Any actions invoked on the UI will be relayed to the component, which will change the state of the actual hardware projector.

After this, another controller is added to the session by the PowerPoint typehandler. This controller provides the user with features for navigating through the slides, using the controller's host device (in this case the user's PDA). During navigation, a small preview of the slide appears on the controllers UI, along with any notes associated with that particular slide. The controller allows the user to gesture with the pen to "draw" over the slide displayed on the projector—the controller receives input events from the PDA and relays those to the backend component running on the projector.

This simple example demonstrates the utility and flexibility of using mobile code to deliver user interfaces, and extend clients with new data transfer capabilities. The PDA shown in Figure 4 not only does not have the PowerPoint application installed on it, but also is running a generic browser application that knows nothing about projectors, slide shows, or PowerPoint. The controller code downloaded from the various components in the session extends the UI behavior of the client dynamically, however.

During this transfer session, there are often occasions when more than one controller is active at a single point in time.

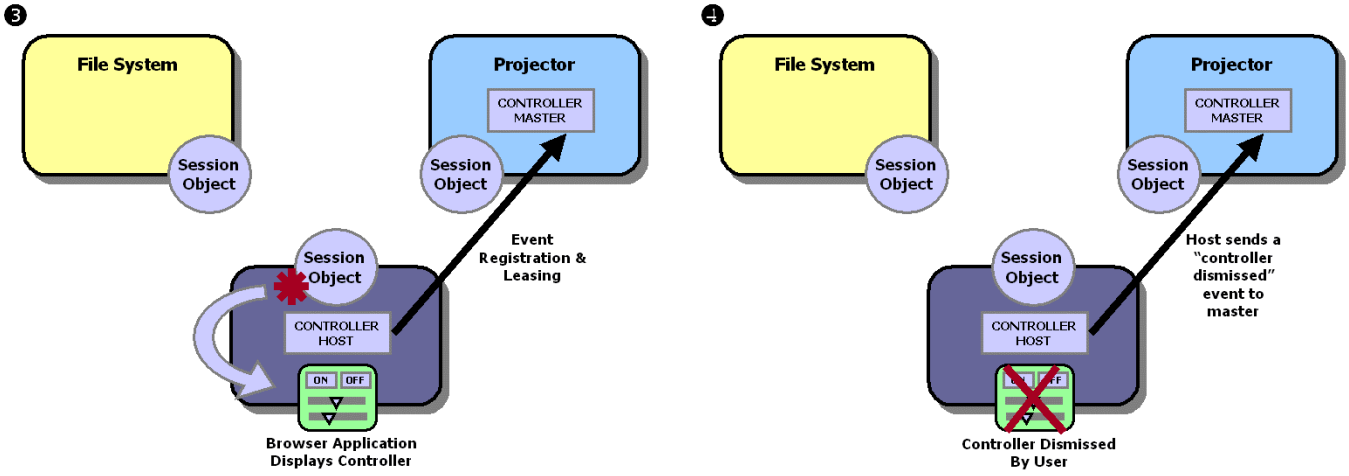

Figure 3: Two subsequent steps in the controller protocol. In 3, one party (in this case the browser) "takes" the controller from the session and displays it. The browser then becomes the "host" for the controller. The host leases the controller from the master, and both arrange to receive events from one another. In 4, Events communicate the state of the controller to the master; leases insure that a failed controller will be noticed by the master. Here, the master receives an event when the user closes the controller.

Figure 4: A PDA displaying the controller for a PowerPoint viewer running on a projector (shown in the background). The PDA in this example knows nothing about PowerPoint.

For a device with a small display it can be problematic to display several controller UIs at the same time. To deal with this issue, the controller toolkit automatically manages controller UIs as they are received by client applications, and automatically places the controls on separate tabbed panels as shown in Figure 5. This mechanism allows users to be aware as controllers come and go, and also provides mechanisms to quickly switch between UIs.

Of course, this particular scenario could have been realized in a number of different ways, either with or without Speakeasy. Numerous solutions to the problem of remotely controlling a presentation have been described in the literature, for example in Pebbles [10]. However, each of these represents a custom solution to this problem and involves specific software to be written for the client platform (PDA or laptop), whereas the Speakeasy solution requires only that a generic Speakeasy-aware browser be installed on the client. All other functionality, such as the ability to control the projector and the slide presentation, is discovered at runtime. Because of this, the same simple browser can be extended at runtime to perform a whole host of other operations that it was not explicitly written to do, such as capture whiteboard images; use the projector to display images, web pages, or MPEG movies; print documents; transfer data from one file system to another; and so on.

Other Components and Controllers

The "travel light" scenario only touches on a small subset of the components and controllers that we have built to date. Other examples include a screen capture component that can export a computer's standard display to any other component on the network (such as a projector); multimedia components that represent speakers, microphone and video cameras. Most of these support their own custom controllers and type handlers.

We have built controllers that can grab keyboard and mouse events from the user's machine and pipe them to a component on the network. The combination of these controllers and a display component allows an application to take complete control of a remote machine's desktop and view it locally.

CONCLUSIONS AND FUTURE WORK

We have presented the Speakeasy UI infrastructure for delivering user interfaces to users in ubiquitous computing environments. This framework not only provides access to user interfaces for services about which the user's application has no prior knowledge, but also allows components in a connection to provide connection-specific user interfaces at precisely the moment when they are needed. Further, the framework provides facilities for providing user interfaces to users on multiple platforms.

We are exploring a number of avenues for future work to address questions that this framework raises. First is the difficulty inherent in creating meaningful and usable aggregate interfaces, when neither the browser nor any of the involved components are expected to have knowledge of each other's semantics.

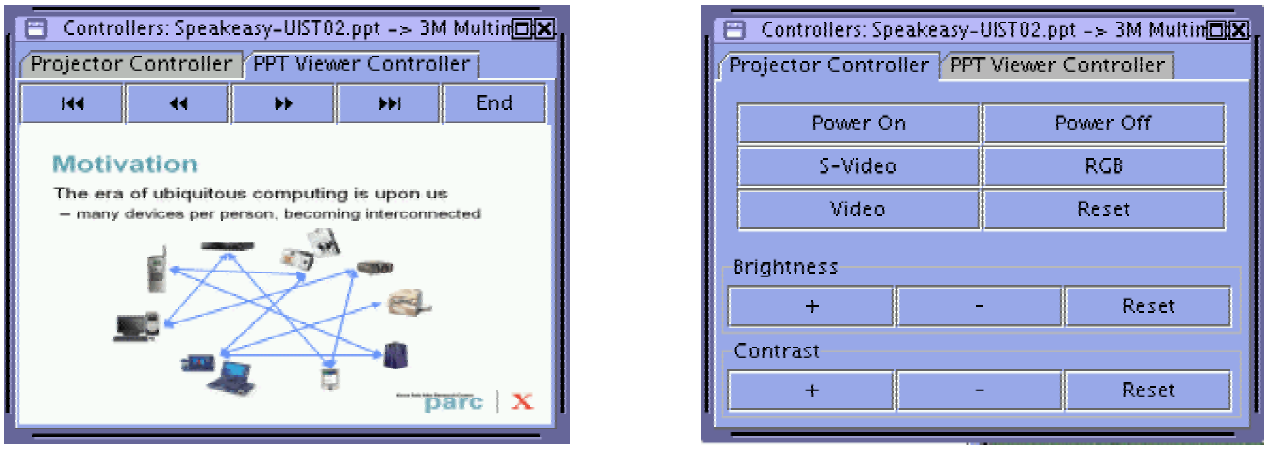

Figure 5: Two controllers for the connection between a remote file and a projector. The controller on the left was added by the PPT Viewer typehandler, and provides controls for the presentation itself. The controller on the right was added by the projector and provides controls for turning the projector on and off, switching the input source, and so on.

In the simple example of displaying a video stream on a projector, for example, a case could be made for *any* of the involved parties adding a controller to pause the video. The approaches we have explored do not address these issues—there is no higher-level "model" of the capabilities of the individual UIs added by the parties in a connection. Model-based user interfaces may hold promise here; others, including ICrafter, have begun to investigate ways to create such aggregate UIs flexibly.

Second, while we have been pleased that the controller framework has proved flexible enough to be used for a range of situations (including relaying a user's mouse and keyboard events over a network), it is not clear to us that controllers are necessarily the right model for such low-level I/O. We have begun to explore new approaches to recombination that allow such low-level I/O streams to be "snapped together," in the same way that our connection framework allows components to be snapped together.

Finally, the controller framework described here illustrates one (and, currently, our only) use of operation-specific user interfaces in Speakeasy. Controllers are the user interfaces to connections, in our model. There are, however, other operations that Speakeasy affords, including discovery of remote components, and examination of contextual metadata on components. We believe that parallel frameworks may exist for these operations also, and have begun to explore these.

REFERENCES

1. Borenstein, N. and Freed, N. *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Messages*. June, 1992.
2. Dennis, J.B. and Horn, E.C.V. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, *9* (3). 1966. 143-155.
3. Edwards, W.K., Newman, M.W., Sedivy, J.Z., Smith, T.F. and Izadi, S., Challenge: Recombinant Computing and the Speakeasy Approach. In *Proceedings of The Eighth ACM International Conference on Mobile Computing and Networking (Mobicom 2002)*, (Atlanta, GA USA, 2002).
4. Eustice, K.F., Lehman, T.J., Morales, A., Munson, M.C., Edlund, S. and Guillen, M. A universal information appliance. *IBM Systems Journal*, *38* (4). 1999. 575-601.
5. Gray, C.G. and Cheriton, D.R., Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of 12th ACM Symposium on Operating Systems Principles (SOSP)*, (1989), 202-210.
6. Harmonia Inc., http://www.uiml.org/specs/uiml2/index.htm.
7. Hodes, T. and Katz, R.H., A Document-based Framework for Internet Application Control. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS '99)*, (Boulder, CO, USA, 1999), 59-70.
8. Kindberg, T. and Barton, J. A Web-based Nomadic Computing System. *Computer Networks*, *35*. 2001. 443-456.
9. Microsoft Corp. *Understanding Universal Plug and Play*. June, 2000. http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc.
10. Myers, B.A., Miller, R.C., Bostwick, B. and Evankovich, C., Extending the Windows Desktop Interface With Connected Handheld Computers. In *Proceedings of 4th USENIX Windows Systems Symposium*, (Seattle, WA, 2000), USENIX Association, 79-88.
11. Newman, M.W., Sedivy, J.Z., Edwards, W.K., Smith, T.F., Marcelo, K., Neuwirth, C.M., Hong, J.I. and Izadi, S., Designing for Serendipity: Supporting End-User Configuration of Ubiquitous Computing Environments. In *Proceedings of Designing Interactive Systems (DIS '02)*, (London, UK, 2002), To Appear.
12. Nichols, J., Informing Automatic Generation of Remote Control Interfaces with Human Designs. In *Proceedings of Conference on Human Factors in Computing (CHI '02) Extended Abstracts (To Appear)*, (Minneapolis, MN, 2002).
13. Olsen, D., Jefferies, S., Nielsen, T., Moyes, W. and Fredrickson, P., Cross-Modal Interaction Using Xweb. In *Proceedings of 13th Annual ACM Symposium on User Interface Software and Technology (UIST '00)*, (San Diego, CA, 2000), 191-200.
14. Olsen, D.R.J., Nielsen, S.T. and Parslow, D., Join and Capture: A Model for Nomadic Interaction. In *Proceedings of 14th Annual ACM Symposium on User Interface Software and Technology*, (Orlando, FL, 2001), 131-140.
15. Ponnekanti, S.R., Lee, B., Fox, A., Hanrahan, P. and Winograd, T., A Service Framework for Ubiquitous Computing Environments. In *Proceedings of Ubicomp 2001*, (Atlanta, GA USA, 2001), 56-75.
16. Venners, B. The ServiceUI API Specification, Version 1.1beta3, 2002.
17. Wakikawa, R., Trevor, J., Schilit, B.N. and Boreczky, J., Roomotes: Ubiquitous room-based remote control from cell phones. In *Proceedings of Human Factors in Computing Systems (CHI '01) Extended Abstracts*, (Seattle, WA, 2001), 239-240.
18. Waldo, J. The Jini Architecture for Network-centric Computing *Communications of the ACM*, 1999, 76-82.
19. Wollrath, A., Riggs, R. and Waldo, J. A Distributed Object Model for the Java System. *USENIX Computing Systems*, *9*. 1996.