Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Int. J. Human-Computer Studies 60 (2004) 666–700

International Journal of
Human-Computer
Studies

www.elsevier.com/locate/ijhcs

Supporting serendipitous integration in mobile computing environments

W. Keith Edwards*, Mark W. Newman, Jana Z. Sedivy, Trevor F. Smith

Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA

Received 26 November 2003; accepted 1 December 2003

Abstract

In the richly networked world of the near future, mobile computing users will be confronted with an ever-expanding array of devices and services accessible in their environments. In such a world, we cannot expect to have available to us specific applications that allow us to accomplish every conceivable combination of devices that we may wish. Instead, we believe that many of our interactions with the network will be characterized by the use of "general purpose" tools that allow us to discover, use, and integrate multiple devices around us. This paper lays out the case for why we believe that so-called "serendipitous integration" is a necessary fact that we will face in mobile computing, and explores a number of design experiments into supporting end user configuration and control of networked environments through general purpose tools. We present an iterative design approach to creating such tools and their user interfaces, discuss our observations about the challenges of designing for such a world, and then explore a number of tools that take differing design approaches to overcoming these challenges. We conclude with a set of reflections on the user experience issues that we believe are inherent in dealing with ad hoc mobile computing in richly networked environments.

1. Introduction

The mobile computing world of the near future is one of both ubiquitous connectivity and ubiquitous computational resources. Users will interact with

*Corresponding author. Tel.: +1-650-812-4405.

E-mail addresses: kedwards@parc.com (W.K. Edwards), mnewman@parc.com (M.W. Newman), sedivy@parc.com (J.Z. Sedivy), tfsmith@parc.com (T.F. Smith).

1071-5819/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ijhcs.2003.12.003

resources such as devices and services around them via omnipresent short-range communications (including both infrastructure-based technologies such as WiFi, and peer-to-peer technologies such as Bluetooth), and also with remote resources via omnipresent long-range communications (such as cellular telephony networks). Some of the resources will be personal, and carried with the user, while others will be embedded in the environment.

We believe that one of the chief challenges that this rich tapestry of interconnectable devices and services raises is that of *serendipitous integration*. *Integration* is the ability for a user to make use of multiple resources in concert with each other. *Serendipitous* integration, then, is the ability to do this in an ad hoc fashion—when, where, and how the user decides.

Why do we believe that this will be one of the primary hurdles to be overcome in mobile computing? Largely, today, integration of devices and services is done through custom software development to provide the "glue" between these resources. This development is often long and arduous, and lags behind the availability of the devices or services themselves. Consider, for example, a Bluetooth adapter bought by a user for a laptop computer. This user may reasonably expect the laptop to now work with Bluetooth printers that she discovers around her. But, while the presence of Bluetooth in both devices allows the laptop to *discover* the presence of such a printer, the laptop will be unable to *use* it without the requisite software needed to integrate it—drivers, management software to allow me to install those drivers, and so forth. While the underlying technology provides some base level of network connectivity, it does not on its own provide seamless integration between two devices that a user may wish to use together. Users must not only wait for the required software to become available, but ensure that it is installed on the various machines that they may wish to use with each other *before the need arises to use them with each other*.

The reliance on custom software arises in many domains. For example, a user can install a universal remote control software package on an infrared-equipped personal digital assistant. But this software is limited in the number of audio/visual devices (makes and models) it can control and, further, likely cannot control devices *other* than stereo equipment in the first place. If the user wishes to play audio from a personal computer through her stereo speakers, even if there is a physical connection between the two, she cannot do so unless the software that is meant to integrate these devices is specifically written to allow this.

The point of these two examples is that, even though we can relatively easily write the software to integrate the devices in question, this does not solve the integration problem in general. Custom solutions to the above problems, even if they existed, do nothing to allow me to perform other tasks, such as view video from my networked digital camera on my personal digital assistant, or allow my digital camera to display a picture on the new wall-sized interactive surface in a meeting room, or any number of other possible combinations that we can easily imagine.

While this lack of easy integration is a problem now (witness the number of remote controls that most households keep easy at hand, for example), we believe it will become an even more onerous problem in the near future. The reason is that there is an ever-expanding range of not just devices, but also *types* of devices. Each new type of device requires custom-built code to allow it to be used with another type of device. Thus, as more and more types of devices appear, there is a combinatorial explosion in the work required to integrate these together.

We believe that this lack of easy integration will increasingly become one of the most visible aspects of interacting with mobile computing technology in the future. This lack of easy integration means that users are unable to "improvisationally" combine computational resources—all means of integration require *some* degree of advance planning (either purchasing devices that are known to work together, packing the cables and adapters needed to allow them to work together, or buying and installing software that allows them to work together). This mismatch between a potential desired combination of resources and the facilities needed to allow them to work together means that users cannot use the resources in their environments in ways that are unforeseen by them, or by the creators of those resources.

Central to our work is the belief that systems should inherently support the ability for users to assemble and combine resources to accomplish whatever work they need to do. This ability to use resources in unexpected ways has been called *appropriation* in the literature (Dourish, in press)—users adopt and adapt technologies to their needs, often in ways unexpected by the developers of those technologies. In the networked world of mobile computing this ability for technologies to be appropriated requires deep support from the computing infrastructure, and also has implications for the sorts of user experiences that can (and should) be created atop them.

To summarize the above points, we believe that such a shift—from relying on pre-built applications to provide tightly bundled aggregations of functionality, to empowering end-users to do such assembly themselves—is crucial in the world of mobile and ubiquitous computing. A set of explorations into how to present such flexibility to users is a key contribution of this paper.

However, there is a second reason we have chosen to explore serendipitous integration: we believe such a reliance on end user assembly is also a *necessary fallout* from new approaches to interoperation. The reason for this is simple: as the network becomes richer and more complex, a device must be able to interact with new types of devices and services on the network, including devices and services created *after* the original device—in other words, types of devices and services it has *not been explicitly programmed to know about*.

Such an arrangement is a departure from traditional approaches to interoperation, which typically rely on the existence of device-specific standards to which other devices are programmed. Without the assumption of such "built in" knowledge about the specific behavior, protocols, and semantics of a peer, users must necessarily take a more active role in arbitrating among potentially interoperable devices on the network. A more in-depth exploration of why we believe this to be the case, as well as an example infrastructure that permits such interoperability without specific device knowledge, is a second key contribution of this paper.

Our work has lead us in two directions. First, we have created an infrastructure, called *Speakeasy*, designed to allow arbitrary, ad hoc integration of devices and services on a network. Speakeasy removes many of the traditional barriers to integration by no longer requiring that applications or devices have specific knowledge of the other types of things with which they will interact. We believe that such an approach—that does not require that specific knowledge about other device types be "pre-built" into each device—is essential for flexible mobile and ubiquitous computing. In such a world, devices may have only "generic" knowledge of each other, but must still be able to interoperate.

Second, we have explored the user experience implications of the ability to freely and improvisationally integrate resources on the network. The primary focus of this paper is on the latter of these two threads, as we explore the implications of devices having only generic knowledge of each other on the user experience. As noted above, we believe that the move towards user interfaces that allow more direct end-user assemblage of functionality will become more important as our networks grow more complex. We have explored a number of paradigms for user interfaces in such a setting, including ways to mitigate several of the problems that arise from such genericity.

There are a number of approaches to integration of networked devices that have been pursued in the past. We believe, however, that none of these can support large-scale serendipitous integration of the sort described above. In the next section we examine the infrastructure requirements that must be in place to allow end-user assembly and use of resources. After this, we briefly describe the systems infrastructure we have built that meets these requirements, and has allowed us to explore serendipitous integration in practice. After this, the major portion of the paper focuses on the user experience of freely combinable networked environments. We examine a number of possible interaction styles that may be used, and describe an iterative design process for one particular user interface. After this, we discuss a number of other user interfaces we have created that embody the lessons we learned in our user studies. Finally, we conclude with a discussion of the lessons we have learned in embarking on this project.

2. Infrastructure requirements for serendipitous integration

In this section we explore the requirements that serendipitous integration places on the computing infrastructure. Many infrastructures have been built that claim "interoperation" of network services as a goal. Systems such as Universal Plug and Play (Microsoft Corp., 2000), and Jini (Waldo, 1999), for instance, all provide the network equivalent of the dial tone—they provide a language for describing the capabilities of services, and for discovering what services are available.

As an example, Universal Plug and Play provides mechanisms for developers to define *device profiles* that describe the capabilities of their devices. Each new type of device would either implement a new device profile, or—if its capabilities closely mirrored those of an existing type of device—use an existing profile. Jini, likewise, allows developers to define the interfaces to their services using the Java language's type definition mechanism. Different types of services implement different interfaces.

But these infrastructures are not in themselves sufficient to support serendipitous integration. The key reason for this is that these systems rely on applications to provide the necessary logic for integrating multiple service types together so that they can use one another. For example, Universal Plug and Play defines two device profiles, called MediaServer and MediaRenderer, which are intended to be used by audio-visual equipment that can produce or accept media, respectively. An example MediaServer might include a video jukebox, while a MediaRenderer might include a flat panel monitor. Custom client applications (called *control points*) are written that provides an "integration layer" atop of these types of devices. These custom clients "know" the operations available on these types of devices, what their semantics are, how to interconnect them, and so forth, and provide a specialized user interface to support integration of these two types of devices.

The problem is that the integration provided by this custom client is entirely specific to the MediaServer and MediaRenderer profiles. Devices that don't implement these profiles are not usable by this client, nor can they be used with existing MediaServers and MediaRenders. The Universal Plug and Play Forum, for example, has defined a Scanner device profile. Despite the fact that scanners produce media (scanned images), this profile has no overlap with the MediaServer profile; allowing the display of scanned images on a MediaRenderer would thus require modifications to the control point software to allow these two device types to be used with each other. This is the case whenever a new type of device is created; the situation is much the same with Jini and other interoperation technologies.

Therefore, a first requirement for serendipitous integration is that the infrastructure must allow applications to control devices, without requiring that those applications be hard coded with the specific details of the devices they control. Why is this? Because if we require that our applications know explicitly about all of the possible things with which they may interact, then we limit them to being able to interact *only* with those things they already know about. New types of devices or services that may appear are inaccessible to them, as in the example of the Universal Plug and Play control point above.

This requirement—that applications are neither expected nor required to have specific knowledge of each and every type of device they may encounter—raises a conundrum, however. In current systems, such as the Universal Plug and Play control point described above, applications largely embody the semantics about how and when to use a device. This semantic knowledge is coded into them by their developers. For example, a "remote control" application for home stereo control "knows" what a television does, that it is appropriate to channel video data from a DVD player to it, and so forth. So it provides access to this functionality through a user interface that makes it easy for users to create these connections, and through explicit programming that can set up the "plumbing" for the desired connection. Put another way, this application encodes the semantics of these media devices to provide users with an easy and situationally appropriate way to access their functionality.

The second requirement for supporting serendipitous integration is that, if applications are no longer expected to embody this sort of semantic knowledge, *end* *users* must be the ones that provide the knowledge of when, whether, and how to use some networked resource. For example, an application designed to support serendipitous integration of networked resources may know nothing about printing, or printers, or half- versus full-duplex, for example. Nor should it be required to, if we expect this application to be able to use resources it is not expressly coded to use. The application *should* however allow the user to make the determination that a particular item is "printable," and instruct the application to connect to and use the printer resource.

While this example may seem foolish—how hard would it be to simply make the application know how to print directly, rather than leaving it up to the user—it illustrates a fundamental shift that must take place for serendipitous integration to become possible. While printers exist today, and are well-supported by most of our software, new types of devices and services are appearing all the time. And, currently, to use these, you must install drivers, new applications, and so forth. (And, all too often, even *then* they are still unlikely to work seamlessly with all types of devices.) And, even in the case of a relatively well-understood type of device, like a printer, new printing protocols and standards are coming along (such as the Universal Plug and Play printer device profile, the Jini printing specification, the Internet Printing Protocol, and so on) that make these newer printers inaccessible to older applications.

In the world we have described so far, users have new opportunities available to them, because they are able to use new types of resources even before application writers have specifically supported them. But these opportunities impose new challenges as well. Now, rather than the constrained interactions that an application writer has decided "makes sense," users have available to them a wealth of networked devices and services, not just the ones innately understood by the applications they use. While this may provide great power, it comes with a cost: how will users make sense of the complex networked landscapes they will inhabit? How will they know that the printer that appears on their screen is, in fact, the printer that they happen to be standing near?

This leads to our third requirement: if we believe that users will shoulder the burden of knowing when and whether to use a certain resource, the infrastructure must explicitly support sensemaking by users. It must allow them to understand what resources are available to them, what its capabilities and characteristics are, and so on.

The next section briefly describes the integration infrastructure we have created to explore the end user questions of serendipitous integration.

3. The Speakeasy infrastructure

The first phase of our research involved the creation of an infrastructure, called Speakeasy, through which we could explore issues around end-user integration. Speakeasy represents an approach to integration that is very different than traditional approaches, such as those embodied by Universal Plug and Play or Jini.

The current version of the system represents approximately four years of refinement and evaluation at PARC, intended to posit new, more scalable approaches to interoperation, and the implications of such approaches on the experience of end users. The system is currently in its fifth version, and runs on a wide range of computing platforms, including personal digital assistants, and Macintosh, Windows, and Linux desktop and laptop computers. A port to a mobile telephony platform is currently in progress.

Speakeasy defines a fixed set of highly generic "meta-interfaces" for interoperation. Unlike Universal Plug and Play device profiles or Jini service descriptions, these meta-interfaces do not define the specific functionality of a service or device. Rather, they define the ways in which that service or device can acquire new behavior, to interact with a previously unknown service or device, and conversely, the ways in which that service or device can itself provide new behavior to its peers. Essentially, they define the axes of extensibility, which allow a new entity on a network to "teach" its peers how to interact with it.

Together, these meta-interfaces allow Speakeasy applications, devices, and services to be extensible along a number of dimensions.

- They allow resources to communicate with each other without having to have specific prior agreement or knowledge about the data communication protocols that either uses.
- They allow resources to exchange data without having to have specific prior agreement or knowledge about the media formats in which that data is encoded.
- They allow end-user applications to present a user interface for an arbitrary resource on the network, without requiring that the application have that user interface "built in" or otherwise know any specific details about the resource.
- Finally, they also support a number of other low-level networking functions, such as the ability to transparently bridge between different networks, perhaps using even different physical media.

These system capabilities are embodied in a number of concepts that the infrastructure exposes to its users. These concepts represent the generic ways that application writers see and interact with the devices and services on the network. The most important of these infrastructure concepts are those of *components*, *connections*, and *context*.

Speakeasy *components* are simply entities that can be connected to and used by other components, as well as by Speakeasy-aware applications. Each component is accessed generically through its meta-interfaces, which allow applications to acquire new behavior required to interact with the component, and also to extend the component's behavior to interact with other components on the network. Using these interfaces, for instance, a component may supply another with a complete implementation of the protocol needed to communicate with it, as well as code for handing media types unknown to the receiver. The component may also provide applications that use it with complete, custom user interfaces that can be used to control the component.

Examples of components include devices such as microphones, cameras, printers, speakers, and displays, or software services such as fileservers and search engines. In total, we have created over two dozen types of such interconnectable components, including all of the ones enumerated in this list as well as others. In some cases, we have used a “proxy” approach to allow devices with only limited computation or communications abilities to be used as Speakeasy components. For example, conference room projectors are “proxied” into the Speakeasy network through the use of a small attached computer, that provides network connectivity for the projector.

The ways in which components extend the functionality of their peers depends entirely on the semantics of the components themselves. A printer, for example, may provide an implementation of the Internet Printing Protocol to clients, to allow them to communicate with it. It may also provide a user interface with various printer-specific options, allowing control over duplex, stapling, and so forth.

The second major concept provided by the infrastructure is that of the *connection*. A connection is simply an association among two or more components for the purposes of transferring data. Such a transfer may represent a personal digital assistant sending data to a printer, or a laptop computer mirroring its display on a projector, or a PowerPoint file being transferred to a projector for display. A connection typically involves a component that is the sender of data, a component that is the receiver of data, and an application that initiates the connection. The initiating application can monitor and control the state of the connection, and thus allow its users to monitor and control the connection.

Finally, *context* is simply structured metadata that is provided by components. Speakeasy imposes no restrictions on the type or use of this metadata; however, it is largely used by components to provide details about their locations, owners, capabilities, and so forth, for human consumption.

A full description of the Speakeasy infrastructure is beyond the scope of this paper (see (Edwards et al., 2002a, b) for more details). Instead, here we focus on our experiences of using this infrastructure to explore the user experience of serendipitous integration, and also how these experiences have been fed back into the design and features of the infrastructure.

The Speakeasy infrastructure does not provide solutions to all of the challenges that arise in a world of serendipitous integration. Still, we believe that our framing of the problem is unique, and that this framing gives us a way to explore issues that are not broached by existing infrastructures, and that will become more and more important in the highly networked, device-rich mobile computing world of the future.

This section has detailed the core concepts that the underlying infrastructure provides. Speakeasy allows interoperation among devices and services that have no specific knowledge of each other’s protocols, media encodings, interfaces, or semantics. Once the core Speakeasy layer is built into a device or service (that is, the device or service has been “componentized”), it can immediately interoperate with any other component already on the network, or that may appear in the future.

Of course, as we noted in the introduction, the fact that components and the applications that use them have only "generic" knowledge of each other—they only "know" that they can interoperate with a peer, not necessarily what it "does"—has important consequences for the user interfaces that can be built with such a system. In particular, creating tightly integrated, holistic applications that use components becomes more difficult—since such applications would rely on being coded against the specific behaviors of components. Instead, we believe that more "generic" applications, without special knowledge of the devices they are controlling, will be common, and must afford the power and flexibility that users expect.

Over the next several sections, we explore a number of ways of mapping these infrastructure features into a suitable and usable user experience. We began by exploring a number of existing approaches to creating "generic" applications for interacting with devices and services. After this, undertook a number of quick, exploratory design and evaluation cycles to reveal early potential problems in the user interface. The point of these cycles was not to produce quantitative data about user performance with our designs, but rather to provide quick, qualitative feedback on them. As a result of these design/evaluation cycles, we have produced a number of applications, built around the user interface concepts refined through our iterations. The needs of these applications have also allowed us to refine the underlying infrastructure in a number of ways.

4. Interaction approaches

There is a large body of work that explores richly networked mobile and ubiquitous computing. Much of this work falls into one of two camps. First is work that explores the user experience of highly interconnected environments. This includes work such as the "personal universal controller" (Nichols et al., 2002), the notion of "information appliances" as a tool for thinking about user experience (Norman, 1998), Mark Weiser's calm technology (Weiser and Brown, 1996), and examples from the space of context aware computing (Abowd, 1999; Kidd et al., 1999).

Much of the work in this category looks at the user experience of specific tasks within a connected world—finding a restaurant in a downtown space, for example. They presume the existence of specific tools designed to support specific user needs, not "general purpose" tools designed to help users cope with the availability of a huge wealth of resources, nor how the users would appropriate and adapt these resources to use them in ways outside those prescribed directly by an application.

The second major category of related work is infrastructure-driven. This is research onto new systems-level facilities designed to make mobile and ubiquitous computing feasible, robust, and workable. These systems tend to focus on enabling technologies such as discovery of resources (Adjie-Winoto et al., 1999), ad hoc networking (Li et al., 2003), control (Microsoft Corp., 2000), and so on. These systems depend on (or presume) the existence of application writers who will combine the various networked resources creating using these infrastructures into well-integrated systems.

In most of these systems, integration is done exclusively through the creation of custom applications, designed explicitly to "know" how to combine and use some select range of service types (again, as in the case of custom-built Universal Plug and Play control points). The most flexible of these systems, such as the Stanford iRoom (Johanson et al., 2002), "lower the bar" to integration to allow experienced administrators to provide the "glue" needed to make disparate systems work together.

We argued earlier for an approach in which the underlying richness and complexity of the network is exposed directly to end users, who are then able to freely control and combine the resources in their surroundings. We believe that this is the only way to allow users to appropriate and adapt the technologies around them in ways that make the most sense for them, rather than the preordained and prescribed ways that application writers may (or may not) have considered and explicitly supported.

The idea of end-user configuration and control is not novel. Systems such as the Macintosh Finder and Windows Explorer support some degree of this: they are end user-oriented, general purpose tools that allow users to configure and control their computing environments. Generic tools like file browsers, for example, allow users to perform a range of operations on their systems—install applications, move or copy data files, explore information stored on the system, access applications through drag-and-drop, and so forth. Other interaction techniques from desktop computing—such as "wizard" interfaces—also allow end user control over some aspect of the computing environment. These tools provide a sequence of screens that ask users questions about their desires, to hone in on the action that the system should take.

But while end user configuration and control has been explored extensively by these general purpose desktop computing tools, doing this in an environment of mobile computing is extremely challenging. The tools described above are designed to work with relatively "closed box" systems—the computing resources they manage are generally slow-to-change and stable over time. Users do not often install new applications that change the built-in capabilities of the system, for example. And wizard-style interfaces, while easy to use, are generally built for supporting a specific task, and provide only a rigid and constrained set of possibilities for the operations they can perform.

In contrast, a "Finder-style" general purpose tool for allowing end users to configure and control their network surroundings has a much more complex set of constraints: it must deal with resources on the network with different, and perhaps unknown, capabilities, it must be able to adapt to resources that come and go as the user moves through different networks, and so on.

Despite this complexity, we still believe that such general purpose tools and interaction mechanisms for finding, using, and combining arbitrary network resources are the best solution for allowing users to access and appropriate the power of the network. Relying on bespoke applications that must be written specially for every possible combination of device types is simply unworkable as the number of device types explodes. In richly networked settings there will *always* be particular combinations of functionality for which no application has been expressly written, and thus end-user integration is an essential aspect to mobile and ubiquitous computing.

There are several models that we explored that have been successfully employed to provide end users with a greater ability to integrate disparate tools together. The Unix pipes system (Newham and Rosenblatt, 1995) is an early, yet powerful, mechanism for allowing command line tools without explicit knowledge of one another to be connected with and used by one another. Pipes are a unidirectional interprocess communication mechanism in which the output of one program can be "piped" to the input of another. Users can form such pipelines of multiple processes through a simple command line syntax. Pipes represent a simple example of a dataflow language, in which users specify how information moves through a successive range of processing steps.

More powerful dataflow tools also exist. These systems allow users to create complex sets of connections among components, and often support feedback loops, decision points, and multiplexing and demultiplexing of connections. In short, they provide many of the features of a full-fledged programming language. These systems often present a graphical "wiring diagram" interaction style to their users. One of the most well-known of these systems is the musical composition application Max (Cycling '74)), which allows users to place music components (such as tone generators and timers) onto a canvas and then connect them together via drag-and-drop gestures. Much like Unix pipes, the interconnection mechanism is independent of the components that are connected—new components can be added to the system and, as long as they provide compatible mechanisms for specifying their inputs and outputs, can be freely connected with existing components.

A yet more powerful approach to end-user integration is through the use of scripting and macros. These systems provide even more power than the dataflow tools described above, but are not as accessible to technically inexperienced users. Perhaps the most common example of a scripting system that allow users to integrate disparate tools together is AppleScript (Perry, 2001), which allows users to "glue together" discrete chunks of functionality from multiple applications. A user can, for example, create a script that programmatically uses the web browser installed on the system to retrieve some content, open it in a document editor, perform some series of operations on it, and then re-upload it to the web. Some of these systems, including AppleScript, allow users to record their actions into scripts that can then be edited and replayed later.

Our goals in exploring these interaction styles were to provide, expose, the underlying flexibility and power of the infrastructure, while ensuring that non-technical users would be able to actually use this flexibility. Although the underlying technical underpinnings of the Speakeasy infrastructure could be "mapped" easily onto a dataflow-oriented model, we rejected this approach as we believed it would require a significant level of technical expertise. Even pipes, a relatively constrained and simple version of dataflow, required that users understand potentially complex chained transformations of data. Our goals were to first see if we could device an interaction style that was amenable to more inexperienced users, and later "fall back" to this model if necessary.

We also rejected scripting for similar reasons. Prior work (such as that by Mackay, 1990) has shown that only savvy users tend to create scripts and macros. (Although these approaches do have the benefit that once created by technically sophisticated users, scripts and macros can be shared and reused by others.)

As our initial interaction mechanisms, we finally settled on using a combination of two distinct approaches. The first of these was a simple drag-and-drop style of interaction for connections between network components. This model is more limiting than pipes, as it allows only pairwise interactions—the dragged component is the "source" of data and the receiving drop target is the "sink." A disadvantage relative to "wiring diagram" styles of interaction is that the connections are only represented transiently; users cannot view a persistent map of connections among components. Still, we believed that this simple model would be easy to understand by our users (because of the widespread familiarity with drag-and-drop style file management systems like the Finder and Windows Explorer), would map relatively well onto our underlying infrastructure, and would be simple to use in a range of client applications.

One additional concern with this approach, however, was that such "raw" access to components might be confusing to users confronted with huge lists of available components. In such a case, the task of combining two disparate components would largely become a search task, since the time spent would be dominated by the time required to find the desired components. Therefore, we also chose a second, parallel interaction mechanism, intended to provide a style of use similar to that of familiar wizards, but with more open-ended functionality. The goal with this interaction mechanism was to provide more task-specific ways for users to make sense of the components available to them, especially given the highly dynamic, heterogeneous, and potentially unfamiliar network settings in which a mobile device may be used. Sensemaking theory (Savolainen, 1993) holds that sensemaking needs cannot be considered in isolation of the situations that create them. This theory inspired us to focus on what users might be trying to accomplish by connecting components in particular situations.

This focus led us to the creation of a notion called *task-oriented templates*, which are persistent descriptions of common tasks that can be created and shared by users. Templates provide a partially specified set of connections, and a set of constraints on the characteristics of components that can be used in those connections. When a user "runs" a template, it presents a wizard-style interface to allow the user to specify the information missing in the template. This mechanism provides the same step-by-step directed interaction style as wizards, but is more adaptable to changes in the set of available components.

As an example, a template named "Give a Presentation" might specify "slots" for a component of type "File" representing the PowerPoint file to be displayed, a component of type "Display" that would receive that file and present it, components that control room lighting, and so on. These slots can be fully specified (for example, the projector slot could be fully specified as "the NEC MT 1030+ projector in room

17”), or more generally specified (“any projector in the building”). At runtime, the wizard interface would allow the user to “fill in” these slots; the constraints on the slots are used to winnow the list of components available to the user to only those appropriate for the given template.

We believed that these templates would fill an important niche in dynamic environments, because they stand in a middle ground between completely unstructured, unconstrained interaction with the resources on the network, and a predefined, potentially inflexible application written by a developer. Templates provide a layer of semantics on top of the raw facilities provided by the infrastructure, and assist in sensemaking by constraining the choices available to the user to only those components appropriate for the task at hand.

After settling on these initial interaction mechanisms, we began the process of designing a general purpose client application, which we called a *browser*, which would embody these interaction mechanisms. This browser application would be the “universal user endpoint,” allowing the user to discover whatever components are available, and then use them with one another to provide virtually open-ended functionality. The next section describes the iterative design of this tool.

5. An iterative design approach

We believed that there was the potential for much confusion with creating a user interface specifically designed to support serendipitous integration. For one thing, people have very limited experience with it—they tend to do those things that are supported by their existing applications. To a large degree, other avenues for integration are simply unavailable to them, especially the non-technically savvy users.

Would users balk at having to understand the semantics of the devices on the network, at having to search for the resources they needed for their jobs? Would they be “conditioned” by existing applications, custom built for certain tasks—and therefore probably easier to use for those tasks?—or would they rejoice at the flexibility with which they could recombine resources on the net?

Because of these fundamental questions, we undertook an iterative approach to designing the user interface. We wanted to better ground our assumptions before “diving in” and expending significant development effort. Iterative design is, obviously, a tried-and-true methodology in human-computer interaction. We do not claim any special methodological contributions in our design and evaluation work; we simply note here that such a co-design approach was particularly useful to us as we set about creating interfaces that would challenge the assumptions of our uses.

The first iteration involved simple paper-and-pencil sketches of the interface, used by the design team to refine the interface concepts. The second involved a higher fidelity mockup, created in HTML and evaluated with potential users. The third involved creation of a browser prototype, designed to be used through a personal digital assistant, and which communicated with “live” components on the network. This prototype was also evaluated. Finally, we took the lessons learned in these iterations to develop three final user interface designs that, while visually distinct, embody the characteristics we believe are essential to support serendipitous integration.

Obviously, the idea of "evaluating serendipity" is somewhat oxymoronic—if you plan to study an event, that event is no longer serendipitous. But while we did evaluate our designs in the context of a number of specific (and, to us, planned) tasks, we were careful to not let such biases slip into the application designs. Our application designs relied only on generic knowledge about the services and devices on the network, of the sort that were provided by the Speakeasy infrastructure, and which would exemplify using such an application to control hitherto unknown devices. This approach allowed us to explore the effects of lack of domain-specific programming in the applications on accomplishing specific tasks in those domains.

5.1. First iteration: low-fidelity mockup

The first iteration in our design cycle focused on validating whether the core concepts in Speakeasy (components, and connections among components) and our initial interaction metaphors (drag and drop for interconnection, and task-oriented templates for reusable connections) were understandable to users and could be conveyed in a simple way on a limited screen space device such as a personal digital assistant. We began this iteration by sketching a number of user interface designs for our browser application on paper (see Fig. 1), to create mockups of its functionality.

In parallel with this sketching exercise, we also articulated a number of integration scenarios that would be representative of the sorts of tasks we expected users to be able to accomplish. These scenarios were meant to tell a concise story about how a user would interact with the technology, and evolved in conjunction with the mockup sketches. As an example, the first, rough version of one of our scenarios was simply stated as, "Give a presentation in the Oasis" (the "Oasis" is the name for a popular conference room at PARC). Once we had a set of tentative user interface concepts in place, however, we could expand the detail of this scenario greatly. The final version was stated as:

> Pat, a manager, is planning to give a presentation in the Oasis later in the afternoon. Pat brings up the Speakeasy browser on a personal digital assistant, goes to a list of templates and chooses "Give a presentation." The template configuration screen appears, and Pat selects "File name: Choose" to bring up a list of components including Pat's file space. After selecting a PowerPoint file, as prescribed by the template, Pat is returned to the template configuration screen with the name of the specified file filled in (as in Fig. 3). Next, Pat selects the slot for the Projector and is shown a list of projectors, arranged by location in the building. Pat selects the projector in the Oasis, at which point slots for controlling the projector and for controlling lights in the room are also filled in with defaults specified by the template.
>
> Having finished configuring the template, Pat names the task "Give a presentation in the Oasis" and places the task on standby. That afternoon, she

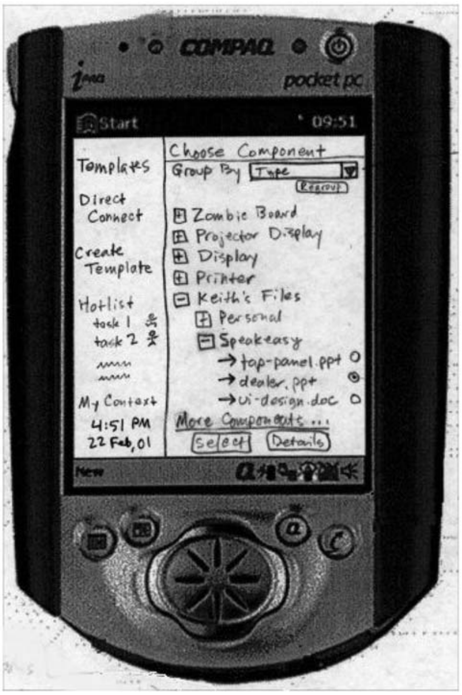


Fig. 1. A low-fidelity mockup sketch of one of the screens of the browser. The sketch was drawn inside an image of a Compaq iPAQ personal digital assistant.

takes her personal digital assistant to the Oasis, and uses the browser to select the task from the list of current tasks and chooses to "Run" it. The controls for the components involved in the task appear in the browser and she is able to dim the lights and control the presentation (e.g. advance to the next slide, return to a previous slide, etc.).

This design divided the personal digital assistant screen real estate into two areas. The first (in the left of the above figure), provided a "toolbar" of sorts for accessing commonly used features, and for changing modes in the application. The second (in the right of the figure) provides an area for selecting components. A combination menu allows components to be grouped by type, by physical location, or by owner.

The screen in the above sketch shows a list of available components grouped by type (such as printer, display, projector, and so on). In cases where multiple instances of a given component type are available, controls are provided to "open" the group, much as in a folder view on a desktop file explorer.

This design supported two styles of interaction by users. First, they could directly interact with any components accessible through the selection area. By selecting a component they could display the user interface associated with that component, for example, or get additional information about that component. They could also directly connect one component to another by first selecting one component and then the other. We called this raw access to component the "direct connection" mode. For example, to use one of the whiteboard capture appliances in PARC conference rooms to take an image of a whiteboard and then print it, the user would first select the desired whiteboard capture appliance, browse again, and then select the desired printer. The underlying infrastructure would take care of the chores of ensuring that these two components communicate with each other at that point.

Second, they could select from a list of available task-oriented templates. These were representations of connections that could be reinstantiated, perhaps while allowing the user to tweak certain parameters of the connection. For example, a "Print the Whiteboard" template might constrain the components that the user sees in the selection area, to make connection easier. This template may first allow the user to specify the printer component and then the whiteboard component, and not display other, irrelevant components. While at this stage of the design, the full functionality and expressivity required by templates was unknown, we believed that at the least this "filtering" ability would be important.

This first iteration was used by the design team to flesh out the user interface concepts, and to make sure that the user interface could, in principle, support the scenarios we had devised. This iteration (which itself consisted of a number of internal "mini-iterations") allowed us to quickly converge toward a design that would warrant more substantial development resources. Importantly, the problems of working with direct manipulation interfaces on small screens led us to focus on techniques such as form filling, rather than mechanisms such as drag and drop.

5.2. *Second iteration: high-fidelity mockup*

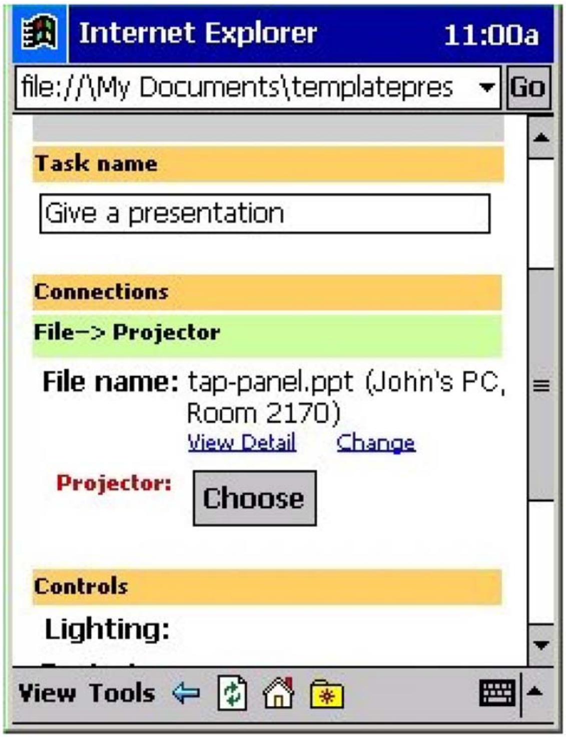
Once our low-fidelity mockups converged to a consistent design, we began work on a more high-fidelity mockup which we would show to potential users. The difficulties in supporting direct manipulation interaction techniques such as drag and drop on a personal digital assistant, discovered in the first iteration, proved to be informative for our choice of platforms for this second mockup. Since the entire user interface could now be supported through the use of forms, we began the creation of a higher-fidelity, interactive mockup in HTML. We simultaneously began constructing the backend functionality of what would become our working browser prototype (the web server framework, and mechanisms for discovering and organizing components, for instance).

The mockup was created as a number of interlinked HTML pages, stored directly on the personal digital assistant. These pages "simulated" an environment populated by dozens of components of various types, and in various locations around our lab. We also created roughly a dozen template pages intended to represent potentially useful tasks (such as giving a presentation, printing a whiteboard capture, and so on). By performing a number of walkthroughs of the design based in our scenarios, we were able to refine the interface design further. For example, during this phase of the design, we realized that we needed to provide a way for users to interact with multiple connections at once. For example, a user might start a connection between a laptop screen and a projector for a demo, but also need to occasionally control a connection between a video cassette recorder and a second monitor. For this reason, we provided controls to allow users to access ongoing connections. We also introduced the terminology *task* to mean a fully specified, running template.

Fig. 2 shows a mockup of the user interacting with tasks and templates. In the example shown here, the user has a text field that can be used to change the name of the task (by default it takes the same name as the template), and also fill in the slots required to run the template (in this screen, for example, the user has the opportunity to select the projector to be used for the presentation; the file has already been specified in this example).

This relatively simple user interface refinement, however, necessitated refinements to the underlying infrastructure to be able to retrieve information about ongoing connections. In the first version of the infrastructure, it was impossible to query a component for the connections to or from it. We added a "connection reflection" facility, to allow the browser to ask components about the ongoing connections of which it was a party.

We also evaluated this mockup with users, who were given a Compaq iPaq personal digital assistant running the Microsoft Pocket PC operating system and the Pocket Internet Explorer application. We asked users to walk through four tasks



Give a presentation

File name: tap-panel.ppt (John's PC, Room 2170)

Projector: Choose

Fig. 2. Configuring a template to prepare it to be run as a task. Users can edit the name to be given to the task, and also select the components specified by the template's slots.

that were representative of our previously created scenarios. The pages created for this mockup were designed to provide a complete path through these tasks, as well as a number of errors that we anticipated users might make.

We performed a lightweight evaluation of this mockup over a two-week period, during which we also continued to refine the mockup based on results of our study. The evaluation consisted of seven subjects, all but one of whom were familiar with personal digital assistant and web technologies. We presented them with a "walk up and use" situation in which they were simply handed a personal digital assistant and the following instructions:

> This Speakeasy browser allows you to connect things together wirelessly. For example, you might connect your personal digital assistant to your stereo in order to control it, or you might connect your laptop to a projector in order to display a file.
>
> You do this by creating *tasks* and running them. To make this easier, common types of tasks have *templates*, but when there's no template, you can also *connect* things *directly*.

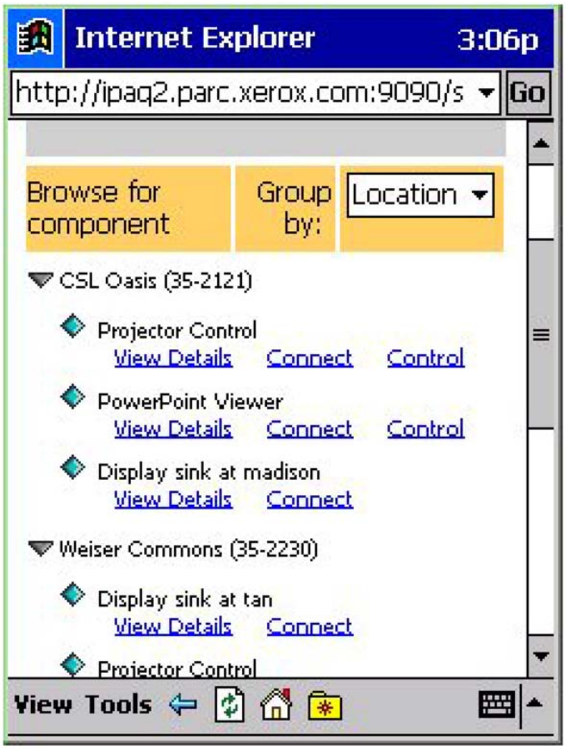
The six participants with personal digital assistant and web experience were able to complete all of the tasks successfully. These participants experienced a number of problems and uncertainties, mostly related to the unfamiliar concepts of "discovering" devices like projectors. Despite these problems, however, post-study interviewing revealed that all six of these participants were able to form a mental model about the concepts exposed by the system and how they worked to allow resources to be assembled together and used. One participant—the one without significant personal digital assistant or web experience—was unable to complete the study.

This round of evaluation confirmed to us the importance of filtering out unwanted components through organizational features in the browser. Users made significant use of the by-type, by-location, and by-owner features. These evaluations also confirmed the value of a task-oriented approach, which was preferred by our participants.

5.3. Third iteration: live prototype

Based on our evaluation of the second mockup, and feedback from users, we proceeded to create a "live" client "browser" application that would be accessed through a personal digital assistant. Because this application was created as an HTML-based web application, we were able to fold in a number of the final design elements from the previous iteration. In addition to creating the client application, we also had to create the actual Speakeasy components needed for the scenarios we planned to evaluate. Fig. 3 shows a screen shot from the final, live browser. This image is of the final design of the component selection screen originally shown in the sketch in Fig. 1.

The evaluation of the live browser was based in the presentation scenario discussed earlier. Unlike that scenario, however, which specifically details the use of task-oriented templates, we provided only vague instructions to participants, that


Internet Explorer 3:06p
http://ipaq2.parc.xerox.com:9090/s Go

Browse for component | Group by: | Location

CSL Oasis (35-2121)
Projector Control
View Details Connect Control
PowerPoint Viewer
View Details Connect Control
Display sink at madison
View Details Connect
Weiser Commons (35-2230)
Display sink at tan
View Details Connect
Projector Control

Fig. 3. An image from the live browser. This image shows available components grouped by location. Note the addition of hyperlinks for accessing descriptive details about the components, connecting a component to another, and for controlling a component.

would allow them to take more open-ended approaches to the task. Thus, rather than explicitly instructing them to use the template as we had before, we simply directed the subjects to present several slides from a recent talk of their choosing. Subjects sent us their slides as Powerpoint files before the study, which we then installed on a Speakeasy "filespace" component that would be accessible on the network during the study.

The six participants in this round of evaluation were all experienced computer users.

Given the set of available components and templates for this study, there were three ways that participants could perform this task. First, they could use an available template called "Give a Presentation in the Commons" (the Commons is another conference room in our lab, and the one in which this study took place). This template was defined so as to constrain the projector that was used to the one in the Commons, meaning that the choice of projector was implicit; users had only to select the file to be presented. Second, they could use a more generic template called "Give a Presentation," which required that users locate and select not only the file to be presented but also the projector. And finally, users could use the direct connection method to locate both the presentation file and the projector from unconstrained lists of all components accessible on the network. Users had to explore the interface to find these templates, as well as the option for direct connection.

All but one of the participants chose to use the first option involving the more specific template, and all were able to complete the task. One of those subjects initially tried the direct connection mode, but quickly gave up and resorted to the specific template once it was found. One participant attempted to use the direct connection mode exclusively and was unable to complete the task without assistance. Once the initial connection setup was made (with assistance in one case), all subjects were able to control the presentation successfully and carry out their talks without further assistance.

5.4. *Observations*

This section highlights some of our observations from the iterative design process, drawing special attention to the differing ways that users tried to address the problems of integration.

The first observation is that users often faced significant confusion when faced with large numbers of components. Even in our live study, in which roughly a dozen components were available, users often had difficulty finding the component that they needed to complete a task. Partially this is due to the small screen space available for displaying components on our personal digital assistant target device. But more importantly, we believe, is that users are simply unaccustomed to dealing with complex search tasks such as our browser presented when doing "simple" chores such as giving a presentation.

This confusion highlights what we believe is one of the key empirical research questions of this work: can we provide enough support to users that they can *actually use* the flexibility inherent in the network? Custom applications work well for allowing users to do what they need to do, precisely because custom applications are designed explicitly to support some particular task. If we believe that, as more and more devices come on the network, the reliance on custom applications will break down, then how closely can we approximate the ease of use of such applications through general purpose tools?

We observed that users' confusion was compounded when the components that users were faced with were irrelevant to the task at hand. Certainly, we had expected that some tools for filtering and organizing components would be important in these designs, but our earliest mockups had not focused on this aspect. After these first sketches, however, it became very clear to us that helping users find the components that they need for a given task would be a significant hurdle, even in the presence of only a few dozen components.

As noted in our earlier discussions, our approaches to eliminating or at least attenuating this confusion included the task-oriented templates mechanism (which would constraint search to a set of "appropriate" components, in those cases for which a template exists), and a filtering mechanism in the browser to support organization by owner, type of component, or location. The fact that users still faced some difficulties finding desired components, even after these measures, highlights the importance of information filtering and some form of context- or task-directed search in these sorts of general purpose tools.

More surprising to us was the range of approaches that users took in trying to accomplish their goals. We intentionally gave users few explicit instructions about the concepts in the system, in an attempt to elicit their thoughts about the way such a system "should" work. For example, one user was focused on using location-based search to find the meeting room in which the presentation was to take place. She believed that once she found and selected the room, everything else would fill in correctly. Another user, an experienced user of Microsoft Office, believed that if she could just "find PowerPoint," she would be able to set up the presentation correctly. Several users focused on "finding the file," and believed that once they had located their file, the rest would fall into place. While the concepts provided by the browser do not precisely map onto these concepts, the organization mechanisms were close enough that all of these users eventually were able to find and use the needed components.

A further problem, which is perhaps clear in retrospect, was that the browser provided only minimal feedback about actions that the user took in the interface. We had overlooked this problem when designing the mockups and prototype system because, in part, we assumed that in a "live" system, users would immediately notice the physical effects of actions taken in the browser. For example, the projector or the lights turning on would be obvious feedback that the action had succeeded, and thus the results would not have to also be presented in the browser.

During our evaluation of the live prototype, however, a number of cases arose in which direct feedback displayed in the browser would have been useful. For example, in one case a user misunderstood the instructions and ended up turning on a projector and displaying their talk in a room other than the one they were in. Since the browser did not tell them what action had been taken, and since they could not observe any effects in the room, they believed that the system had simply failed.

In other cases, even when a user selected the right projector, there were problems with feedback because the projector would take a few moments to warm up. During this time, some users were not sure whether they had performed the correct action to turn on the projector. (We should note, however, that the remote control normally used to turn on the projector does not fare any better in this regard.) Better feedback from the browser would have mitigated some of these problems.

One final issue, which we were aware of but became more problematic as we realized the need for feedback, and especially feedback of error conditions, is that web-based applications are notoriously bad for displaying asynchronous notifications. Our browser, since it was based on off-the-shelf web technology, could not regularly pull updates from the server without annoying redraw problems. The server, likewise, could not easily push updates to the client. Again, during our initial design we were aware of these limitations, but believed that they would not be overly detrimental to the browser design. As we progressed through the evaluations, however, it became clear to us that the infrastructure must provide a way to deliver constant feedback to users; further, this feedback may need to be delivered asynchronously, especially in the case of errors or other exceptional conditions.

6. Final system designs

The iterative process described above allowed us to quickly identify problem areas in our design. Some of these problem areas—such as dealing with large numbers of components—are inherent in our vision of serendipitous integration. Others, however, are artifacts of the web-based technology we used—such as the lack of ability to dynamically update the client application.

Since this first browser application, we have created a number of other designs that embody the lessons learned from our observations. Much like the browser just described, each of these is a result of an iterative process of design and development. These applications are all visually distinct, and most are intended to support situations and contexts different from the "generic browser" situation envisioned by our first prototype. Still, however, each of these applications must cope with the same challenges outlined above.

In the next sections, we briefly discuss these three client applications. We use our observations from the discussion of our first browser, and the challenges inherent in serendipitous integration, to motivate the solutions that these applications take to overcoming these challenges.

6.1. Supporting collaboration: Casca

Casca is a Speakeasy application intended to support informal collaboration among workgroups. The application allows users to create *converspaces* ("conversation spaces"), which are collections of networked resources that can be accessed and used by any user who is a *member* of the converspace. The application is meant to allow multiple, ongoing collaborations to coexist, each in its own converspace (Edwards et al., 2002a, b).

The notion of "spaces" for collaboration is not new—systems as diverse as chat rooms, Groove (Groove Networks Inc., 2002), Lotus Notes (Kawell et al., 1992), GroupKit (Roseman and Greenberg, 1992), and so on embody similar notions. Unlike many of these collaborative tools, however, Casca allows sharing not just of certain pre-defined types of content, such as text or applets. Instead, Casca converspaces can contain any component that can be located over the network.

Using Casca, a user can create a new converspace and invite other users to become members of it. Any member can then invite other members, or add components to the converspace. Once added to the converspace, the component is both visible and accessible to the other members of the converspace.

Fig. 4 shows the Casca main window. The large area to the right is a tabbed list of converspaces. Below the selected converspace is displayed icons representing the users who are members of that converspace. In the center of the application is a pane that, in essence, replicates the "selection" area of the browser application we have

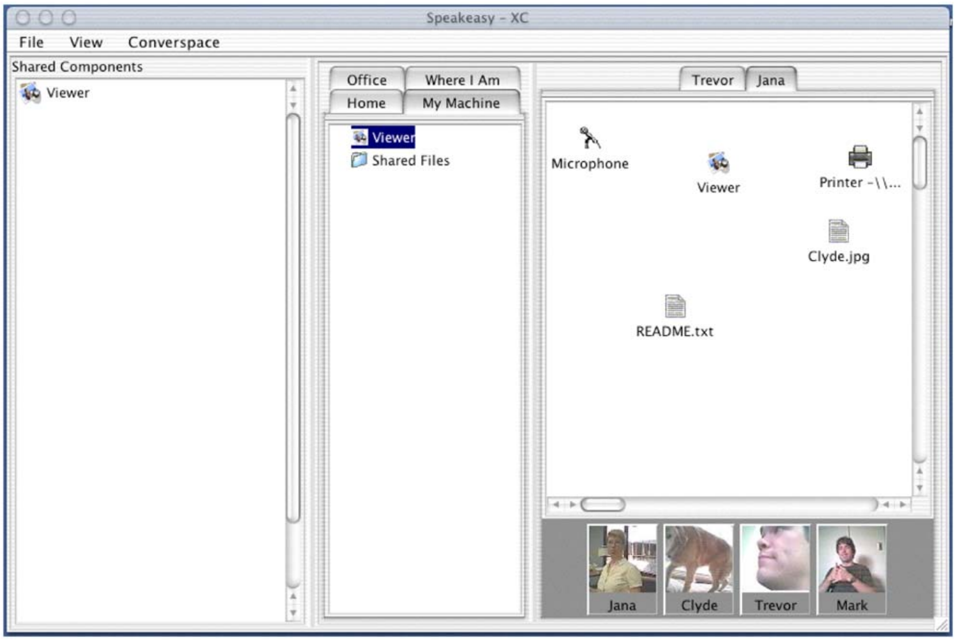


Fig. 4. The Casca main window. The large area to the right contains components that have been added to the converspace; the icons below represent the current members of that converspace (these icons can be clicked on to determine if a user is currently online, or to send mail or an instant message to a user). The middle area is used for component selection, much as in the first browser. In this case, the tabs represent pre-defined organizations through which to view the components available on the network. Note the "My Machine" tab, which displays only the components running on the same machine as Casca. The area to the left simply shows all of the components that the user is currently sharing with other members of the converspace.

just discussed. A Casca user can use this pane to organize and view the set of components accessible from his or her local machine. These components can then be dragged into the converspace to share them with other users.

For example, a user may drag a component representing a file from her local file system into the converspace; this immediately makes it visible and accessible to all other members of the converspace. Likewise, users can drag in components representing resources such as printers (allowing a remote collaborator to send materials to my home printer, for example), or video cameras (to set up an improvised video conference, for instance).

The design of this application has benefited from the lessons learned during our earlier iterative design process, and provides new solutions to some of the challenges enumerated above. The first is the notion of *explicit scoping*. For most of the time that a user is interacting with Casca, her focus is on the main canvas area that represents the set of components available to members of the collaboration. Unlike the open-ended set of components presented by the first browser, the components visible here are constrained to the set that has been *explicitly* added to the converspace by a member, presumably because of the salience of the component to the collaboration.

In essence, this design allows a group of human collaborators to share filtering decisions to winnow down the set of available components that they must commonly deal with. By adding a component to the converspace, a user makes it immediately visible to others, even if that component would "normally" be difficult to locate in a browser like the one described previously. Of course, the user that adds the component must still locate it, using much the same mechanisms as those provided by the first browser. But once located and added, it becomes easily accessible to the other collaborators without their having to know the details of how it was found.

Casca is intended to be run on a machine such as a laptop, with greater screen space and computational capacity than the personal digital assistant, which was the target of the first browser. Thus, the machine hosting the Casca application may itself also potentially host Speakeasy components. For example, the file system on the laptop may be exposed as a Speakeasy component (allowing it to transfer data to and from arbitrary other components on the network), as may be hardware and attached devices on the laptop, such as its screen, input devices, speakers, and so on.

Because of this intended use, the selection area in the left of the Casca main window provides a grouping category for components on "My Machine" (see the figure above). This group includes all of the local components executing on the same host as the Casca application itself. Casca is the first application we have built which explicitly supports this notion of "local" components as a separate category that may be useful for search and organization.

In addition to explicit scoping, Casca uses the notion of the converspace to accomplish another goal which was not addressed at all by the first browser. This is the problem of *access control* of components on the network. By default, local components are configured to not be accessible over the network by other components; they can only be interconnected with other components running locally on the same machine. But as components are added to a converspace, their access control parameters are modified to allow them to be interconnected with components owned by the other members of that converspace. This design choice conflates the user action required to indicate intended visibility of a component with that intended to set its access control permissions. In effect, we "reuse" this action, which the user would have to take anyway, to hide the explicit setting of access control rights. The pane at the far left of the application in the above figure represents all of the components that the user is sharing with others; this is meant to provide a constant reminder of the user's access control state with respect to other collaborators.

The converspace notion has proved to be a useful design construct that not only overcomes some of the problems of "component overload" observed by the first browser design, but leverages this construct to support other user needs, such as access control.

6.2. A second generation general purpose browser: Nexus

Another follow-on client application is called *Nexus*. This application is essentially a descendant of the browser described above—it is intended to provide general purpose access to all of the components available on a network. It is designed to run on personal digital assistants, but also works on more traditional computers, such as common laptops and desktops. Fig. 5 shows the application running on a handheld computer.

With the creation of more domain-specific tools (for collaboration, such as Casca described above, and home audio/visual control, such as the tool described below), the need for general purpose tools is somewhat mitigated. Thus, we have designed Nexus to be a "power user" tool, which provides access to the raw functionality of the Speakeasy network, allowing arbitrary recombination of the resources discovered there.

The first browser prototype introduced the concept of task-oriented templates as a "middle ground" between purpose built applications and direction connections among components. In this first browser, however, templates were neither easily editable nor configurable; they could only be modified by an administrator. The Nexus application, however, supports a more full-featured and mature use of templates. Knowledgeable end-users can create templates as XML files that are loaded by the applications and used to recreate configurations of components.

Nexus also extends the basic organizational capabilities of the first browser. Whereas the first browser was largely limited to organizing components based on their type, location, or owner, Nexus provides a sophisticated query language that allows users to create arbitrary categories through which to view the network. While common queries are provided that mirror those in the first browser (based on owner, for example), power users can create new queries that allow them to view the network in custom ways.

The system can also be configured with more intelligent location based browsing. For example, the system can be configured with maps of the current environment that allow component locations to be visually plotted. When coupled with

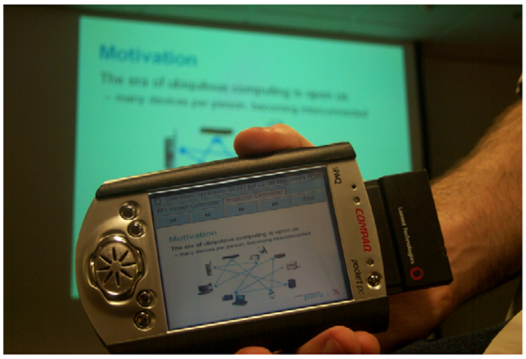
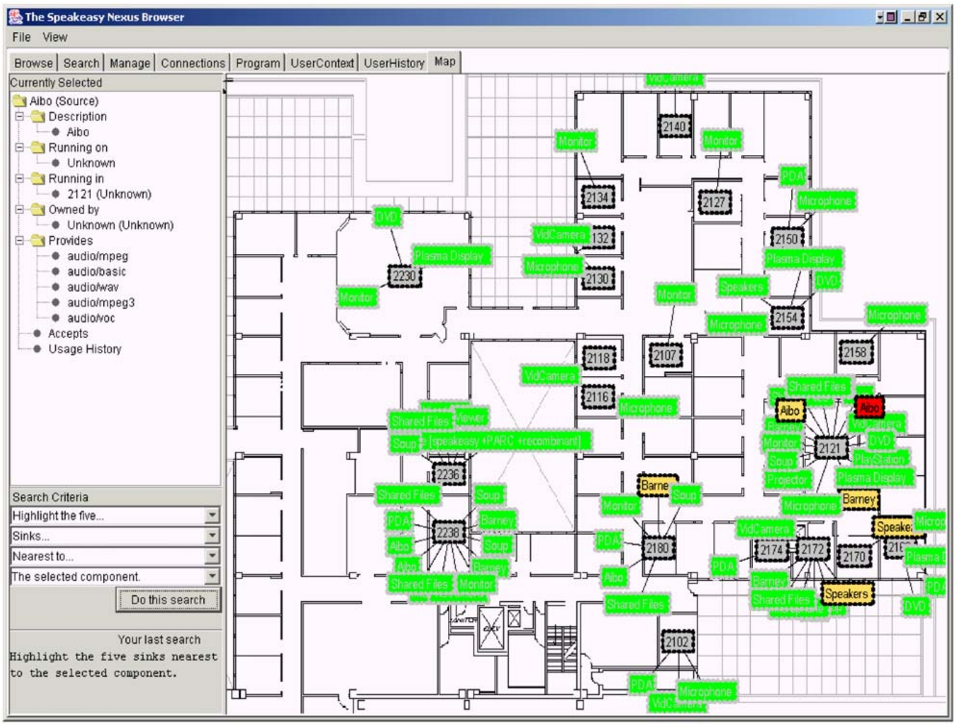


Fig. 5. Nexus running on a handheld computer to control a PowerPoint presentation.



Currently Selected
- Aibo (Source)
  - Description
    - Aibo
  - Running on
    - Unknown
  - Running in
    - 2121 (Unknown)
  - Owned by
    - Unknown (Unknown)
  - Provides
    - audio/mpeg
    - audio/basic
    - audio/wav
    - audio/mpeg3
    - audio/voc
  - Accepts
  - Usage History

Search Criteria
Highlight the five...
Sinks...
Nearest to...
The selected component.

Your last search
```
Highlight the five sinks nearest
to the selected component.
```

Fig. 6. Location-based search facilities in Nexus. The image shows the floorplan of a work area. Rooms with components are highlighted. The search criteria panel to the lower left allows the user to find components based on location.

information about the physical space, the browser can also organize components by notions such as "Closest Printer" for example. Fig. 6 shows an example of this mapping capability. In this image, Nexus is running on a laptop computer. The tabs at the top give access to connection management, templates, and history- and location-based search for components.

Perhaps most importantly, Nexus moves away from the web-based infrastructure of the first browser. This has both good and bad points. On the positive side, much richer user interfaces are possible in a non-HTML environment. Nexus is created using Java, and so very rich user interfaces are possible; further, these interfaces can take better advantage of the extensibility provided by the Speakeasy framework to extend the Nexus applications to entirely new user interface behaviors (some of these are discussed below). On the negative side, a client-resident application requires explicit installation of software on the device. The web-based browser, on the other hand, requires only a browser, which is likely to be already installed.

On the balance, though, the benefits of the richer infrastructure outweigh the costs in most situations. Moving away from HTML allows the personal digital assistant to play a more direct role in participating in the interaction with components. For example, one of the downsides of the first implementation arose purely from the fact of its web-based implementation. Web-based interfaces are notoriously poor at handling asynchronous notifications from the server about changes in user interface state. So the first browser was poor at handling unexpected error conditions that occurred asynchronously during a connection. Essentially the browser was coded to "know" when to ask for a user interface; if some condition occurred that necessitated the display of a user interface at any other time, the browser could not cope with this demand.

This constraint eventually led to a more robust user interface delivery architecture for the Speakeasy platform as a whole. Our explorations with the first browser prototype led to the creation of a notion known as *controllers*, which are user interface elements that are pushed asynchronously from components to a browser, at a time of the component's choosing. See (Newman et al., 2002) for more details on the particulars of this mechanism.

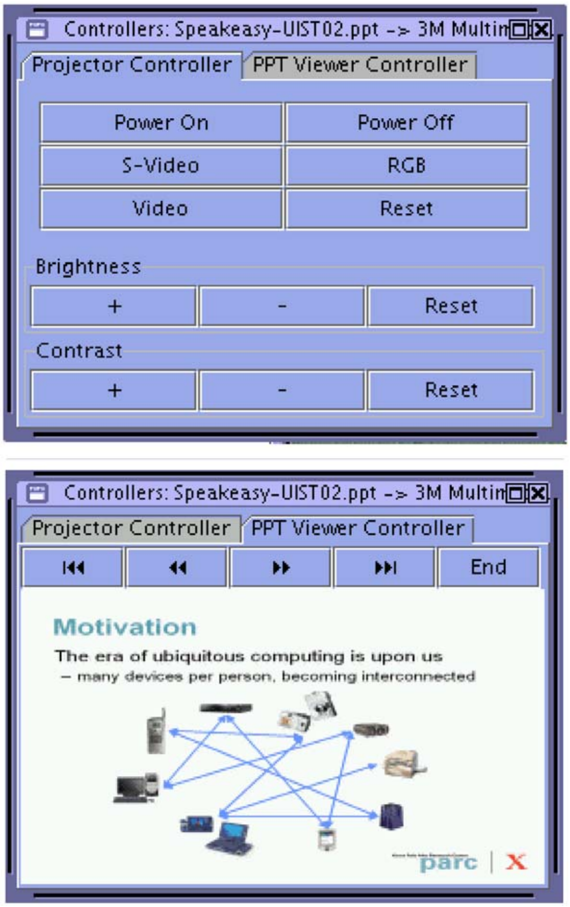
Nexus was our first application to take advantage of this new capability in the infrastructure. Fig. 7 shows an example of the controller mechanism in action. Here, Nexus is being used to control a PowerPoint presentation on a projector, much as in the scenario described earlier. The image on the top shows the controls for the projector, while the one on the bottom shows the controls for the PowerPoint presentation itself. Note that in this case, Nexus has been extended with the ability to not only move through the slideshow, but also to display miniature versions of the slides in the talk, even though PowerPoint is not itself installed locally on the personal digital assistant. Further, users can draw with the stylus over the slide to mark on the screen. The browser itself need know nothing about PowerPoint, projectors, or drawing in order to accomplish this behavior. Instead, it is delivered dynamically to the browser at the time it is needed.

6.3. *An audio/visual centric client: Mediahub*

The final application we describe here is intended for more specific usage scenarios than our other applications, which are fairly "generic." This applications is a home *mediahub* application that might be run on a networked settop box, to provide access to audio/visual facilities on the home network.

Our implementation of this application was targeted at a small form-factor personal computer intended to be hooked up to a television screen and speakers, and operated via an infrared remote control. In addition to the mediahub application itself, which provides the ability to integrate and control components on the network, the settop box also includes a number of local components. These include, in our implementation, a media library component (used to store digital audio or digital video clips), a DVD component (used to access DVD media in the internal drive), a display component (which can accept media from any component on the network and present it on the television monitor attached to the settop's video output port), speaker components (which likewise accept audio data and render them to attached speakers), a television tuner component, and so forth.

One could certainly imagine a very general purpose interface to such a box, much as the one presented by our first browser, and by Nexus. We believed, however, that



Motivation

The era of ubiquitous computing is upon us
– many devices per person, becoming interconnected

Fig. 7. Two examples of dynamic controllers, running in Nexus. The image on the top shows the projector's controller while the one on the bottom shows the controller for the PowerPoint presentation itself. Note that Nexus assembles all of the relevant controllers for a connection into a group of tabbed panes.

an application designed to be used in a fairly constrained setting should appropriately have a more constrained user interfaces, even though the possibility would certainly exist of using the user interface to control arbitrary components on the home network.

Therefore, the mediahub uses a number of "built in" templates that provide easy access to common functions: "Watch a DVD," "Listen to Music," "Watch Television," and so forth. While these templates do not use the same XML specification as the ones used in the Nexus application, they are similar in intent: they provide task-oriented ways to easily achieve certain combinations of functionality. In many cases, one or more of the components specified by the templates are implicitly local components hosted on the settop box. Fig. 8 shows two

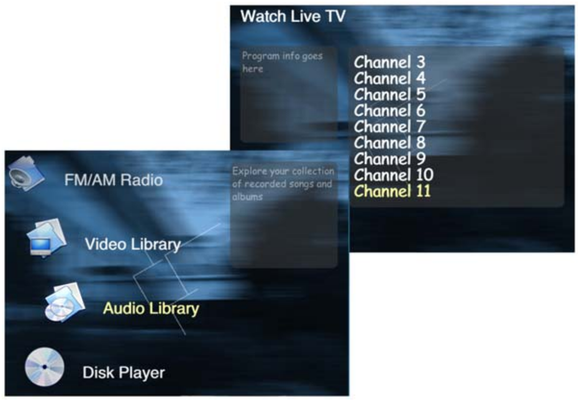


Fig. 8. The mediahub application. The image on the lower left shows some of the main menu items in the user interface. Each selection represents a built in template of functionality. The image on the upper right shows a template-specific user interface, in this case, for tuning channels on a television tuner component.

screens of the mediahub; the menu items on the main screen are common repeatable operations.

Thus, the "Watch Television" template means to connect the internal television tuner component to the internal display component, causing the live television stream to be displayed on the monitor. This approach circumvents many of the problems of working in the presence of abundant components that characterized our evaluations of the first browser prototype. By understanding the common operations that users may wish to perform in a given setting, we can provide easier access to those.

The mediahub does not just allow interconnections among only local components, however. The user interface allows the common operations to be "specialized" for remote use. For example, when the user selects the "Watch a DVD" template, the next screen allows the user to make a selection between the local display component, and other components that may be found throughout the home. This functionality keeps the task-oriented filtering that obviates many of the problems with the presence of large numbers of components, yet still allows the selection of arbitrary components where appropriate.

This interaction style—allow the user to select only the items that are appropriate in a given situation—is reminiscent of the "Direct Combination" work of Holland and Oppenheim, (1999). In that work, users essentially narrow a large search space of options by constraining some aspect of an operation.

7. Reflections

A key motivating question for our work was the issue of how to effectively design user interfaces that would inherently support "serendipitous" discovery of available resources and co-adaptation of tasks. While we believe that we have taken significant steps in this direction, the very nature of the goal makes it difficult to evaluate. Indeed, by their very nature it is difficult to dictate "serendipitous tasks" to test subjects. Rather, these types of interactions arise out of natural, day-to-day contact with the technology.

Nevertheless, the deployment and testing of the Speakeasy browser provided us with important information regarding the engineering concepts of serendipitous integration and towards user interaction issues in our environment, which we have tried to apply to the design of new tools.

We believe (and participants in both studies agreed) that we ended up with a usable browser as well as a substantial list of additional possible improvements. However, we further believe that some of our experiences have implications beyond the specific browser application described in this paper. As mentioned before, we believe that general purpose browser-style applications that allow users to discover and interact with arbitrary devices and services in an ad hoc fashion and will play an important role in future mobile computing environments. In the absence of specialized applications for every conceivable task, a more generic tool—one without specialized domain knowledge—will necessarily play a part. In this section, we explore some potential implications of our work on this wider class of applications.

7.1. Co-design is a powerful approach for exploratory systems

First, and at a very high level, we believe that co-design of the infrastructure and interface can be a powerful way to ensure that the concepts in the interface can be mapped onto features in the infrastructure, and that those infrastructure features provide the underpinnings necessary for a useful and usable interface. To a large degree, the mechanisms in the infrastructure suggest or even determine what concepts can be exposed in a user interface; and, likewise, the desired user interface may dictate certain properties that must be present in the infrastructure.

We consider our iterative co-design process to be a lightweight way to refine both aspects of the system in tandem, as suggested by (Edwards et al., 2003). Prototyping and experimentation of with the user interface led to new infrastructure features, including the connection reflection and controller mechanisms discussed earlier.

Obviously, co-design is not a new approach. But we believe its use is especially imperative in situations where either (or both) of the infrastructure and resulting interfaces are radical departures from norms about what users will accept.

7.2. Information filtering is essential

The ability to easily filter unwanted components is essential in a general purpose tool designed for use in richly networked environments. As the number and range of devices increases, integration tasks quickly become search tasks, and hence their usefulness will in large degree depend on the ability of tools to support efficient search and organization.

Our early designs supported a fixed number of organizations that we believed would be useful—organizing by type or component, location, and owner. Later tools expanded this range categorization features to allow custom organization. Still, however, an interesting research question remains: what aspects of components are most useful for search and organization, and for what tasks and types of components? This is an important question, because these attributes must be reflected in the information provided by components (and thus must be addressed by the infrastructure), as well as in the constraints provided by template builders.

Our later tools expanded on the basic organization capabilities of the first browser in another important way. Tools such as Nexus can take the *user's* context into account to provide a more dynamic view of the components on the network. So, for example, instead of seeing a list of room numbers with the components in them, Nexus can provide a list of components in the *same* room as the user, if such information is available to the browser (through either location sensing technologies, or explicit information from the user). While we have not yet evaluated this aspect of the interface, we believe that such a dynamic approach to information filtering, in which the organization presented to the user is tailored to the user's location, history, and tasks, could prove useful. Likewise, Casca's distinction between local and remote components, seems to be an essential and salient for users to understand what resources *they* are providing into a networked environment.

Another approach, less complex but still useful, is to use the Direct Connection interaction mechanism to dynamically filter the components in a connection to only compatible ones. This approach does not require explicitly created templates, but can still greatly reduce the search space for desired components.

7.3. Task-oriented templates provide semantic filtering

Task-oriented approaches to component access provided great benefits in our experiments. Users experienced far greater success in accomplishing tasks when they used templates than when they did not. We believe that the primary reason for this is that templates help in sensemaking, since they inform the user about what sorts of components would be appropriate for any given task. Further, the wizard-style interaction supported by templates provides essentially a semantic filtering mechanism—users do not deal with all of the complexity of the network, because they only see relevant components.

The benefits of such a task-oriented approach are perhaps not surprising: task-oriented approaches to presenting information are commonly found in instruction documentation, for example, where they have been found to improve the productivity of users over feature-oriented documentation (Odescalchi, 1986).

The key challenge, in an infrastructure such as ours, is providing a task-oriented user interface at all, given the fact that the client applications are not required to understand component semantics. Currently, the Speakeasy infrastructure requires that such task-oriented templates be created by hand, by a user with specific knowledge of the task at hand. Once created, however, they can be reused by others;

in essence, this approach allows the user interface to leverage the knowledge and work of a human user to provide a more directed experience to other users.

In some sense, templates serve as a "bridge" between dealing with raw connections and using bespoke applications. They provide a lightweight way to capture domain semantics—what components are appropriate for a given task, what constraints are salient for them, how they should be connected—without requiring full-fledged programming on the part of a developer, or scripting on the part of a user.

7.4. Appropriation is enhanced when multiple mechanisms are supported

As we noted in our observations, different people use different approaches in how to integrate resources, even when carrying out the same task. Certainly part of this is because our browser presents a novel set of opportunities for most people, who thus have little prior experience of such end user integration. But we also believe that users will naturally take different approaches to solving any given problem. As we saw, some users focused on location, others on data, and still others on applications.

The ability to take use multiple mechanisms to solve a problem is important in a general purpose tool designed to allow users to appropriate resources around them. It allows them to use the "best" mechanism for the task at hand—whether that's best because of their prior experiences or expectations, or best in some objective sense.

The multiple organizational techniques used by the first browser showed how users' ability to appropriate technologies is enhanced through supporting multiple mechanisms. Later tools enhanced this by adding still more mechanisms, such as the map-based location features in Nexus.

7.5. Redundant feedback is essential

In all of our evaluations, it was clear that the lack of feedback in the browser about the results of users' actions was problematic. Users lacked confidence in the results of their actions, not only when the effects of their actions were invisible, but even when those effects could, in principle, be observed locally. The latter might occur when there is a failure in some other part of the system (for example, a service failure or disconnected cable), or because the effects of an action are not immediately apparent (such as a projector being slow to warm up).

While so-called "invisible interfaces" are espoused by some in the ubiquitous computing community, we feel that users in such environments will need *more* tangible feedback and control, not less. For example, being able to ascertain what components are currently doing is valuable to users attempting to discover whether certain components are available for their use.

Two features in the underlying interface—connection, reflection and controllers—were specifically added to provide this sort of feedback about system behaviors. These mechanisms provide feedback about both individual components, and about interactions involving multiple components.

We believe that effective, continual feedback about the state of the network and the components in it is essential for the sorts of tools described here. In mobile computing settings, components may come and go, failures may occur, and the state of the world may change rapidly. The infrastructure, and the applications built atop it, must be prepared to deal with such dynamism. Such fluidity may also suggest that "pull"-oriented user interface technologies (such as the web) may be inappropriate at accommodating such change.

7.6. Tiny laptop or big remote control?

There was quite a bit of variability among the participants in terms of their expectations about the degree of automation provided by the browser and/or environment. For example some users, after having selected the file for their template and "run" the resulting task, expected to have to turn on the projector manually and actively searched within the browser for the projector controls. Other users expected that the projector would be automatically turned on for them and set to the correct input, and were utterly mystified when this did not happen. One member of this latter group thought that if he could just "open" his file, everything would just work. This sentiment was similar to that expressed by the participant in the mockup study who said, "I just want to choose one button and have it work." Tantalizingly, these expectations of greater automation seemed to come more readily from participants with less computer programming background, perhaps suggesting that users who are less familiar with the limitations of technology were more inclined to have higher expectations about its capabilities.

Another way of explaining these differences, however, is that users' attitudes might have been conditioned by their perception of what this particular application running on this particular device was most like. The users who expected more automation seemed to also follow a model of how they would perform the task using a desktop or a laptop computer. They expected that once they found their file, they could just "open it" as one would do by double-clicking in the Windows Explorer or the Macintosh Finder. On the other hand, users who expected less automation seemed to regard the browser as something more like a sophisticated remote control, where the main advantage of the browser was that it provided convenient access to controls for the various devices and services. Upon reflection, we realized that the latter model is more akin to what we, the designers, had in mind, but the former model is also quite compelling and we plan to investigate adopting some aspects of it in the next version of the browser.

The challenge here is to understand, and hopefully even exploit, users' expectations about the affordances of an environment, based on the device they use to interact with that environment. Different models are possible, and may be more or less advantageous in certain circumstances.

8. Conclusions

We have described a possible future in which arbitrary devices and services can be interconnected and used without specific prior knowledge of one another. We believe that such a world will bring with it a host of questions about what user actions can—or should—be supported: how will users discover the devices and services that are around them, and how will they organize, understand, and ultimately use these devices and services to accomplish some task?

A fundamental question we have posed is whether and how users will adapt to interfaces that present only generic means for interacting with services and devices on a network. We began with an initial, iterative design approach to help us to understand how users would approach such a mode of interaction. Even though exploratory, this series of experiments nonetheless yielded interesting results, about how users conceptualize interactions on the network, and several ways in which the "gap" between generic representations of devices and services, and more task-specific representations, can be closed.

We have adapted the results of these studies into the systems described later in the paper, to explore additional design approaches to dealing with the challenges inherent in working in richly networked worlds. We believe that these tools help point the way toward further improvements in not only the user experience, but also in the underlying Speakeasy infrastructure.

References

Abowd, G.D., 1999. Classroom 2000: an experiment with the instrumentation of a living educational environment. IBM Systems Journal 38 (4), 508–530.

Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., Lilley, J., 1999. The design and implementation of an intentional naming system. In: Proceedings of ACM Symposium on Operating Systems Principles (SOSP).

Dourish, P. The appropriation of interactive technologies: some lessons from placeless documents. Journal of Computer-Supported Cooperative Work, in press.

Edwards, W.K., Newman, M.W., Sedivy, J.Z., Smith, T.F., Balfanz, D., Smetters, D.K., Wong, H.C., Izadi, S., 2002a. Using Speakeasy for Ad Hoc Peer-to-Peer Collaboration. In: Proceedings of ACM Conference on Computer-Supported Cooperative Work, pp. 256–265.

Edwards, W.K., Newman, M.W., Sedivy, J.Z., Smith, T.F., Izadi, S., 2002b. Challenge: recombinant computing and the speakeasy approach. In: Proceedings of he Eighth ACM International Conference on Mobile Computing and Networking (Mobicom 2002).

Edwards, W.K., Bellotti, V., Dey, A.K., Newman, M.W., 2003. Stuck in the middle: the challenges of user-centered design and evaluation for infrastructure. In: Proceedings of ACM Conference on Human Factors in Computing Systems (CHI), pp. 297–304.

Groove Networks Inc., 2002. http://www.groove.net/.

Holland, S., Oppenheim, D. (1999). Direct combination. In: Proceedings of ACM Conference on Human Factors in Computing Systems (CHI), pp. 262–269.

Johanson, B., Fox, A., Winograd, T., 2002. The interactive workspaces project: experiences with ubiquitous computing rooms. IEEE Pervasive Computing 1 (2), 71–78.

Kawell, L., Beckhardt, S., Halvorsen, T., Ozzie, R., Greif, I., 1992. Replicated Document Management in a Group Communication System. In: Marca, D., Bock, G. (Eds.), Groupware: Software for Computer-Supported Cooperative Work, IEEE Computer Society Press, pp. 226–235.

Kidd, C., Orr, R.J., Abowd, G.D., Atkeson, C.G., Essa, I.A., MacIntyre, B., Mynatt, E.D., Starner, T.E., Newstetter, W., 1999. The aware home: a living laboratory for ubiquitous computing research. In: Proceedings of Second International Workshop on Cooperative Buildings (CoBuild'99).

Li, Z., Li, B., Xu, D., Zhou, X., 2003. iFlow: middleware-assisted rendezvous-based information access for mobile ad hoc applications. In: Proceedings of International Conference on Mobile Systems, Applications, and Services (MobiSys), pp. 71–84.

Mackay, W.E., 1990. Patterns of sharing customizable software. In: Proceedings of Conference on Computer Supported Cooperative Work '90.

Microsoft Corp., 2000. Understanding Universal Plug and Play.

Newham, C., Rosenblatt, B., 1995. Learning the Bash Shell. O'Reilly and Associates, Sebastopol, CA.

Newman, M.W., Izadi, S., Edwards, W.K., Smith, T.F., Sedivy, J.Z., 2002. User interfaces when and where they are needed: an infrastructure for recombinant computing. In: Proceedings of Symposium on User Interface Software and Technology (UIST).

Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Pignol, M., 2002. Generating remote control interfaces for complex appliances. In: Proceedings of ACM Symposium on User Interface Software and Technology (UIST), pp. 161–170.

Norman, D., 1998. The Invisible Computer. MIT Press, Cambridge.

Odescalchi, E.K., 1986. Productivity gain attained by task-oriented information. In: Proceedings of 33rd International Technical Communication Conference.

Perry, B.W., 2001. AppleScript in a Nutshell. O'Reilly and Associates, Sebastopol, CA.

Roseman, M., Greenberg, S., 1992. GROUPKIT: A groupware toolkit for building real-time conferencing applications. In: Proceedings of Conference on Computer-Supported Cooperative Work (CSCW).

Savolainen, R., 1993. The sense-making theory: reviewing the interests of a user-centered approach to information seeking and use. Information Processing and Management 29, 13–28.

Waldo, J., 1999. The Jini Architecture for Network-centric Computing. Communications of the ACM, 76–82.

Weiser, M., Brown, J.S., 1996. The Coming Age of Calm Technology. http://www.ubiq.com/hypertext/weiser/acmfuture2endnote.htm.