Experiences with Recombinant Computing: Exploring Ad Hoc Interoperability in Evolving Digital Networks

W. KEITH EDWARDS
Georgia Institute of Technology
MARK W. NEWMAN
University of Michigan
and
JANA Z. SEDIVY and TREVOR F. SMITH
Palo Alto Research Center

3

This article describes an infrastructure that supports the creation of interoperable systems while requiring only limited prior agreements about the specific forms of communication between these systems. Conceptually, our approach uses a set of "meta-interfaces"—agreements on how to exchange new behaviors necessary to *achieve compatibility at runtime*, rather than requiring that communication specifics be *built in at development time*—to allow devices on the network to interact with one another. While this approach to interoperability can remove many of the system-imposed constraints that prevent fluid, ad hoc use of devices now, it imposes its own limitations on the user experience of systems that use it. Most importantly, since devices may be expected to work with peers about which they have no detailed semantic knowledge, it is impossible to achieve the sort of tight semantic integration that can be obtained using other approaches today, despite the fact that these other approaches limit interoperability. Instead, under our model, users must be tasked with performing the sense-making and semantic arbitration necessary to determine how any set of devices will be used together. This article describes the motivation and details of our infrastructure, its implications on the user experience, and our experience in creating, deploying, and using applications built with it over a period of several years.

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*User-centered design; Prototyping; Interaction styles; Theory and methods; Interaction Styles*; H.1.2 [**Models and Principles**]: User/Machine Systems—*Human Factors*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.2.12 [**Software Engineering**]: Interoperability—*Distributed objects*

General Terms: Human Factors, Design, Standardization

Authors' addresses: W. K. Edwards, School of Interactive Computing & GVU Center, Georgia Institute of Technology; email: keith@cc.gatech.edu; M. W. Newman, School of Information, University of Michigan; J. Z. Sedivy and T. F. Smith, Computer Science Lab, Palo Alto Research Center.

DOI 10.1145/1502800.1502803 http://doi.acm.org/10.1145/1502800.1502803

Additional Key Words and Phrases: Mobile code, discovery, infrastructure, recombinant computing, Obje, Speakeasy, interoperability, ubiquitous computing, user interfaces

ACM Reference Format:

Edwards, W. K., Newman, M. W., Sedivy, J. Z., and Smith, T. F. 2009. Experiences with recombinant computing: Exploring ad hoc interoperability in evolving digital networks. ACM Trans. Comput.-Hum. Interact., 16, 1, Article 3 (April 2009), 44 pages. DOI = 10.1145/1502800.1502803 http://doi.acm.org/10.1145/1502800.1502803

1. INTRODUCTION

Many scenarios of mobile, pervasive, and ubiquitous computing envision a world rich in interconnectable devices and services, all working together to help us in our everyday lives. Indeed, we are already seeing an explosion of new, networked devices and services being deployed in workplaces, the home, and in public spaces. And yet, in reality it is difficult to achieve the sorts of seamless interoperation among them that is envisioned by these scenarios. How much more difficult will interoperation become when our world is populated not only with *more* devices and services, but also with *more types* of devices and services?

These visions of ubiquitous computing raise serious architectural questions: how will new devices and services be able to interact with one another? Must we standardize on a common, universal set of protocols that all parties are required to agree upon? Will only the devices and services[1] that have software explicitly written to use one another be able to interoperate, while others remain isolated? How will such networks be able to evolve to accommodate new devices that may not have even existed at the time current devices were created?

These architectural questions in turn pose pressing issues for the user experience of ubiquitous computing. Architectures that impose such constraints, that prevent us from fluidly adapting and using the technology around us for the purpose at hand, cannot support the sort of "calm technology" envisioned by Weiser and Brown [1996]. Instead, they are likely to bring the frustrations of software incompatibility, driver updates, communication problems, and version mismatches—in other words, isolated islands of interoperability, with few bridges among them.

As we explain shortly, these weaknesses are inherent in current architectural approaches to device-to-device communication; overcoming them requires new approaches in which devices can interact with new types of peers *without having to know about them ahead of time.* Such an approach would allow devices to work with one another without requiring replacement, upgrade, or patching, allowing networks to evolve to accommodate entirely new types of devices easily. In other words, such an approach would remove the system-imposed constraints on interoperability that prevent us from freely combining and using the technology around us. If, on the other hand, the software in our devices must be updated to work with every possible new type of thing they may encounter, then we will be

[1]While we use the terms *device* and *service* here and elsewhere in the article, in practice there is little distinction between a purely software service executing on some host on the network and a hardware device on the network. The arguments here apply generally to any system that can be used by another across a network or other connection.

locked into a world of limited interoperability, and the requirement for lockstep upgrades of the devices on our networks.

This article describes work aimed at overcoming the user experience burdens that result from current approaches to device-to-device communication, particularly when those approaches are "scaled-up" to large numbers of device types. Among the main contributions of this article are a new approach to interoperability, designed to support *ad hoc interoperation* among devices on a network, and an exploration of the human interaction implications of this new approach. The key benefit of our approach is that it can allow interoperation among devices that are not expressly built to work with one another, thus allowing the network to evolve to compatibly accommodate entirely new types of devices, without modification to existing devices on the network. As we shall describe, this approach overcomes some of the traditional human burdens associated with device interoperability, albeit at a cost: although our approach can allow devices to work with previously unknown peers without driver installation or upgrade, it cannot support the rich, seamless interactions among devices that are possible when they have been purpose-built to work with each other.

The technical aspects of our work have been embodied in a system called *Obje* (previously called *Speakeasy*) [Edwards et al. 2002a], which is a platform we have built, refined, and lived with for over five years, and which provides a set of mechanisms for ad hoc device discovery, extensibility to new application-layer protocols and data type-handling capabilities, and a security model that supports decentralized authorization and access control. This article provides a rationale for our approach to interoperation, the user experience implications of this approach, a description of the Obje infrastructure, and a discussion of our experiences using this technology to explore issues of interoperability in the face of evolution.

In the next section of this article, we explore the foundations of communications among devices on a network, and how assumptions implicit in these foundations point to the need for new models of interoperation and user interaction. After this, we examine our design rationale, which takes the form of a set of premises we call *recombinant computing*, and which we believe can support the forms of interoperation we argue for in this article. These premises include the use of generic, fixed interfaces to guarantee compatibility; mobile code to allow dynamic extensibility; and the necessity that users be involved in dictating how devices will interact with one another. We then contrast our approach to others in the research literature, and highlight the interaction trade-offs (as well as architectural trade-offs) inherent in different approaches to interoperability. Next, we describe the core Obje infrastructure itself. As noted earlier, Obje has been used and refined over a period of approximately five years, and the current implementation reflects our experiences and the lessons we have learned in using the system. We conclude with a detailed description of some of these experiences and lessons.

2. PARADIGMS OF COMMUNICATION

Fundamentally, communication between any two systems depends on prior agreement about the *interfaces* supported by those systems. These take the

form of the protocols, operations, data types, and semantics that, ultimately, two systems must be coded against in order to work with each other. For example, to interact with a service, a client must be explicitly written to "understand" the service's interface—what operations are available, how to invoke them, and what the semantics of these operations are. In the case of the web, for instance, there is agreement on the form of communication (HTTP), the format of data exchanged (HTML, mainly), and the semantics of that data (most clients will issue a GET request when they encounter an IMG tag, for instance). In the case of Universal Plug'n'Play (UPnP) [Jeronimo and Weast 2003], these agreements include not just the generalities of UPnP itself (such as the SOAP protocol, and the requirement that device descriptions be expressed in XML), but also the *device-type-specific profile* used by a particular UPnP device, which defines the set of allowable operations for that device type. For a client to be able to interoperate with a UPnP digital music jukebox, for example, the client must have software that is coded to use the UPnP MediaServer profile, which is specific to those types of devices that can store and transport media files.

Current approaches to interoperability take what might be termed an *ontological* approach: a standard[2] is created that defines how a device's functionality is exposed to and accessed by peers on the network. Knowledge of this standard—in the form of software that implements its required protocols, data formats, and semantics—is then built into peer devices on the network. New types of devices, which do not sufficiently resemble the existing ontology to the point that they can be retrofitted into it, necessitate the creation of yet more interface definitions and the expansion of the ontology of device types.

This same approach is taken by virtually all networked communication systems today. UPnP defines device interfaces, termed profiles, for a range of device types, including scanners, printers, media devices, HVAC systems, and so forth. Bluetooth likewise defines similar profiles for headsets, telephones, hands-free car systems, and so on. Virtually without exception, new versions of these standards define new device interfaces (or revise existing device interfaces) that will be unknown to existing devices on the network; this means that existing devices—with only knowledge of the previous set of device types—cannot interoperate with any new types of devices defined by the expanded ontology.

Arrangements such as these *gain interoperability at the expense of evolution*. Because the agreements necessary for communication must be built in at development time to all communicating parties, the network cannot easily take advantage of entirely new types of devices. New types of devices must either be retrofitted into the existing ontology in order to maintain backwards compatibility (at the cost of losing access to whatever specialized functionality the new device type provides), or new interfaces must be created that describe the new device type (at the cost of having to *update all existing devices* with

[2]We use the term "standard" loosely, to mean knowledge that is required for communication and that is agreed upon by both communicating parties. These agreements can be codified formally by a *de jure* standards body, codified informally by *de facto* open standards, or may even represent "private" protocols, known only to the two devices that are communicating.

the programming to communicate with the interface used by the new device type, or, more likely, by simply replacing the existing devices with newer ones).

Such models of communication seem fundamentally at odds with a vision of ubiquitous computing predicated on technological abundance, in which new types of devices are easily deployed, integrated into the environment, and appropriated by users.

3. RECOMBINANT COMPUTING: MOVING FROM DEVELOPMENT-TIME TO RUNTIME AGREEMENTS

If communication requires such up-front agreement, then the central question posed by our research is as follows: is there a way to reframe the agreements necessary for communication in a way that can better support evolution to accommodate arbitrary future device types? Further, can this be done in a way that allows for a rich and useful user experience, without the hassle and incompatibility imposed by current approaches?

Our project has been exploring an approach we call *recombinant computing* that shifts some of the knowledge necessary for communication from development-time to run-time. It does so by relying on agreement only on a minimal set of development-time interfaces that are then used to allow devices to negotiate further necessary agreements, along with the behavior needed to implement these, at runtime. The term "recombinant computing" is meant to evoke a sense that devices and services can be arbitrarily combined with each other, and used by each other, without any prior planning or coding beyond this minimal set of development-time agreements.

In order to achieve interoperability in this model, three criteria must be met for the base-level agreements that are built into devices at development time. The first is that the interfaces that devices support must be *fixed*, since this is what guarantees future compatibility. If the interfaces necessary for communication are allowed to evolve, then existing devices may be unable to communicate with devices built using the later versions of the interfaces. Second, the interfaces must be *minimal*. Otherwise they are unlikely to be adopted and implemented fully by developers, imposing new barriers to interoperability. Finally, such a fixed and minimal set of interfaces will necessarily be *generic*, since any small fixed set of interfaces cannot capture all possible device-specific notions, for all unforeseen types of devices.

The problem, however, is that relying only on a small set of static, generic interfaces is incredibly limiting, and supports only simplistic interactions among devices. For example, although one could achieve interoperability by dictating that all devices on the network, no matter what their function or semantics, exclusively use plain text over FTP as their communication mechanism, such an arrangement clearly limits the range of possible applications.

Therefore, our system couples these static development-time interfaces with the runtime exchange of new capabilities over the network. In essence, the fixed agreements become *meta-interfaces* that, rather than dictating how two devices interact with each other, instead dictate the ways in which the *devices can acquire new behavior that allows them to interact with each other*. This

new behavior takes the form of mobile code that is provided by devices on the network to their peers at the time of interaction.

The approach is reminiscent of Kiczales et al.'s metaobject protocols [Kiczales et al. 1991], in which a set of fixed interfaces can be used to provide runtime extensibility (in Kiczales et al.'s case, of programming languages). Here, however, new behaviors are provided over a network connection in the form of mobile code, rather than simply across a procedure call boundary.

Using this model, devices only build in the most generic agreements (the interfaces for acquiring and invoking mobile code), and defer other agreements necessary for interoperation until runtime. When a new device appears on the network, it provides certain behaviors to existing peers to bring them into compatibility with it, essentially "teaching" its peers how to interact with it.

We believe that this approach is qualitatively different than one based on requiring detailed *a priori* agreements about all aspects of a device's syntax and semantics. Approaches based on extensive and changing ontologies place the burden of work inappropriately: when a new type of device appears, *every other* device on the network must be updated in order to work with it. In an approach based on runtime agreements, in which functionality needed for interoperation with a new device is provided by the device itself, the burden of work is borne only by the new device.

3.1 The Necessity of User-Supplied Semantics

Ad hoc interoperability poses important implications for the user experience. Under current ontological approaches, if a device can interoperate with a peer at all, one can reasonably expect that the device "understands" the semantics of that peer—what it does, when to use it, and so forth. A Bluetooth phone, for example, "knows" what a Bluetooth headset is capable of and when to use it (when a phone call happens, connect to the headset and stream audio between it and the phone) because this understanding of how to seamlessly mesh the semantics of the two devices has been built into them by their developers, allowing rich and semantically-informed interactions between them.

If devices can only interact with the peers with which they are expressly programmed to interact, then new types of devices will be inaccessible to them. Such is not the case in a world of ad hoc interoperability, however. Instead, the *common case* will be that devices will encounter new types of peers about which they have no special semantic knowledge: they will be able to communicate with such devices, yet without necessarily knowing what the new device *does*.

For example, under a model of ad hoc interoperability, a phone may detect that there's a device present that it can communicate with, and perhaps send audio to, but may not be expected to know whether that device is a headset, a speaker in a public place, a storage device, or a service for transcribing speech to written text. Nor arguably *should* the phone have to have such knowledge in order to communicate with the new device since, if interoperability is our goal, we do not want to require that devices only be able to interact with things they already know about.

If devices do not contain the specialized programming necessary to allow them to work in semantically-informed ways with specific types of peers, then it is incumbent upon the *user* to provide the semantic interpretation and arbitration missing in the devices' programming. In the example above, it must be the user who is tasked with understanding what a particular device does, when it makes sense to use it, and so forth; the role of the infrastructure, under this approach, is simply to *allow* the interaction to take place, should the user decide to do it.

This is an example of what Fox and Kindberg have termed "crossing the semantic Rubicon," requiring users to take on some of the burden of semantic interpretation that once was the domain of applications [Kindberg and Fox 2002].

We believe that this implication that users must be "in the loop" in determining when and how to use newly encountered devices, is inherent in any model of ad hoc interoperation, and it is central to the design of Obje. Users have the role of providing the semantic interpretation that may be missing in the programming of the application. This requirement has ramifications on the sorts of user experiences that we can create, which in turn has implications for the design of the infrastructure that must allow users to easily understand and use the resources around them.

4. RELATED WORK

There are a number of systems that address issues similar to those addressed by Obje, both in the general goals of interoperability and in the specifics of the individual approaches.

Many systems, including traditional remote procedure call systems (such as CORBA [Object Management Group 1995]), as well as the newer Web Services standards (such as SOAP [Box et al. 2000], provide a substrate for communication among networked services; essentially, they provide a framework on top of which new application-layer interfaces can be created, including interface definition languages, building block protocol formats, standards for parameter marshaling and unmarshaling, and so forth. All of them leave the task of defining the actual service interfaces up to developers, with little focus on standardizing the service interfaces themselves. The result is a large number of service-specific interfaces, each of which must be known to their clients in order for those clients to be able to use them. These systems fundamentally address a different problem than the one addressed by Obje: they are focused on creating common formats and toolkits to support the easier creation of networked services, rather than on ensuring interoperability among arbitrary services.

Other systems, such as Universal Plug and Play [Jeronimo and Weast 2003], take interoperability a step higher up the stack by defining not just low-level protocol formats, but also a standard set of device interfaces. These systems exemplify the ontological approach to interoperability described earlier: they achieve compatibility by requiring agreement on an expanding coterie of standard device interfaces. When a new device type appears, it necessitates the

creation of a new standard interface that describes, and allows access to, the device's functionality; peers must be written against this new standard interface in order to use the new type of device. In the case of UPnP, devices and applications are coded against *device profiles* that specify the standard operations supported by a given device type (see UPnP Forum [2005]). Even if the software on a device is written to allow it to interact with UPnP MediaServers, for example, it would still have to be recoded to work with MediaRenders, Printers, Scanners, and any additional new type of profile that comes along. Further, the profiles are themselves subject to change, meaning that new versions of the profiles may introduce new features that are inaccessible to older devices. A device built to use version 1.0 of the MediaRenderer specification cannot take advantage of features defined in the 1.1 version, for example.

The approach taken by UPnP and other ontological systems (including Bluetooth [Bluetooth Consortium 2001], USB [Universal Serial Bus Implementer's Forum 2000], and others) exemplifies the state of practice in interoperability today. These systems achieve compatibility, but at the expense of evolution: the creation of a new type of device requires additions or modifications to an existing family of standards; these standards are often slow to evolve, and device support lags behind even the standards. Finally, once the standardization process has been completed and the necessary new devices built, deploying these new devices onto the network requires the update of *every existing* device on the network in order for them to work with the new device.

Technologies such as Sun's Jini [Waldo 1999] provide a greater degree of insulation between communicating devices on a network. Whereas systems such as UPnP require agreement on both device profiles and underlying network protocols (such as SOAP over HTTP), Jini requires agreement only on the *abstract* interfaces exposed by devices and services. In Jini, clients are written against a service's interface, described as a Java interface type; the service, then, provides a service-specific *implementation* of this interface at runtime in the form of mobile code that is downloaded to the client and executed by it. The advantage of this approach is that Jini can allow clients to communicate with services using protocols not previously built into the clients, since services provide to clients the code necessary to use them. Still however, like UPnP, Jini requires that clients be written against specific service interfaces. Jini defines standard interfaces for a number of core infrastructure services (such as the Jini Lookup service), but other domain-specific service interfaces are not defined by Jini itself.

Jini's use of mobile code makes it similar to the Obje approach architecturally. For example, Jini allows service-specific, mobile code-based user interfaces to be downloaded by clients [Venners 2005]. The capabilities in Jini make it a platform on top of which one could define a set of generic meta-interfaces, similar to those defined by Obje; Jini itself does not specify such interfaces however. Also, Jini is also closely tied to the Java language and platform, requiring Java as a common mobile code execution format; as described later, Obje supports a range of mobile code formats.

Other systems have explored mobile code-based approaches to distributed computing, although often not in the context of supporting evolution while

maintaining interoperability. Bharat and Cardelli's work [1995] on migratory applications, for instance, uses mobile code as a way for an agent to move across the network to resume on a new host. While that work shares a common mechanism with Obje—the use of mobile code—it does not share a common goal. Rather than supporting interoperability, Bharat and Cardelli's work is largely focused on enabling applications to move seamlessly through the network, carrying data and program state as they move. These differences in aims drive differences in mechanisms: in Bharat and Cardelli's model, code *moves* to a new host and begins executing in its own thread. In Obje, by contrast, small fragments of code are *copied* to a peer, where they are only invoked as necessary through calls into known methods on that code by the receiving application. Bharat and Cardelli's agents cease executing on old hosts as they move to a new destination. In Obje, transfer of code to one peer does not preclude transfer of code to other peers; in other words, execution of transferred code need not cease at one point simply because an Obje device is interacting with a second peer. In Bharat and Cardelli's work, agent code is not expected to be called into from the local computing environment on its new host, except by an "agent server" that knows how to load and execute arbitrary agents. In Obje, mobile code is expected to implement interfaces known to the receiving device so that code on that device can directly invoke and interact with code received over the network.

HP's Cooltown project [Kindberg and Barton 2001] is concerned with extending web presence to devices and services situated in the physical world and is, in many ways, close to Obje in its philosophy for interoperation. Cooltown leverages web standards—the ubiquity of HTTP, and the ability to use it as a small, fixed, and generic interface to a range of services and devices—to achieve ubiquitous interoperation. But by relying on web standards, Cooltown is also bound by the web's limitations. These include reliance on the data types and protocols commonly used in the web, the lack of easy dynamic extensibility, and a focus on browser-server style interactions, rather than interactions between services initiated by a third party. Olsen's [2000] XWeb system presents what is essentially a web-oriented architecture, but with extensive modifications to remove many of the limitations of the web that affect systems like Cooltown: much like Obje, XWeb aims to move away from "interactionally impoverished" protocols and data types such as HTTP and HTML. XWeb provides a new transport protocol, called XTP, that provides rich mechanisms for finding, browsing, and modifying information represented in hierarchical form on a remote service or device. On top of this base protocol, XWeb provides high-level abstractions (called *interactors* and *XViews*) for modifying server data, either at an atomic level or in aggregate. The XWeb approach, by moving away from some of the underlying technical limitations of the web, can provide much richer interaction, including easy adaptability to a range of input devices, easy linking between back-end data and front-end interfaces (including notifications when back-end state changes), and so forth. In comparison to Obje, however, XWeb still lacks the easy dynamic extensibility afforded by mobile-code based approaches, including extensibility to new data transport protocols, data types, and discovery protocols.

The iRoom [Johanson et al. 2002] and Appliance Data Services [Huang et al. 2001] projects at Stanford provide ad hoc interoperation of loosely coupled services by allowing them to share data through tuplespaces (a concept first described by Gelernter [1985] in which arbitrary data tuples can be written onto a blackboard, and detected and consumed by other services). Such systems can provide far greater flexibility than agreement on fixed protocols, since the set of tuple types is easily evolvable; new tuple formats can coexist with old ones, for example. For two tuplespace-based services to work together, however, they must still have prior agreement on the format of the tuples; the extent of interoperability is dictated by the extent of a common language of tuple syntax and semantics. Of course, Obje also dictates a common set of interfaces that we expect to be known to all parties, but our intent is for the interface set to be fixed and not open-ended as in the case of many tuplespace-based systems.

Still others have focused on the problem not just of interoperation but of preserving interoperation in the face of evolution. For example, Ponnekanti and Fox [2004] highlight the range of compatibility problems that can result from the evolution of web services. Their solution to allowing evolution while preserving interoperability depends on static and dynamic analysis tools to validate compatibility among services, along with automatically generated middleware components that can bridge between otherwise incompatible versions of services. In comparison to the work of Ponnekanti and Fox, which is largely aimed at mediating compatibility issues for existing WSDL-based web services, Obje takes a more "clean slate" approach, abandoning the existing web services-oriented architecture for one that can potentially support evolution without the need for custom middleware generators or other support infrastructure on the network.

Another project that focuses on interoperable evolution is HydroJ, a set of language extensions to Java intended to allow the easy creation of interoperable web services [Lee et al. 2003]. The creators of this system note the problems caused by evolution in traditional RPC-based systems, in particular what they call the "brittle parameter problem": that any change to the parameters of a remote interface breaks compatibility. HydroJ takes a position reminiscent of the iRoom, relying on semistructured data to mitigate some of the brittleness associated with traditional RPC systems. This approach permits variation in the contents of messages; much like in tuplespaces, portions of a data structure that are not understood are simply ignored by recipients. In comparison with Obje, HydroJ has similar goals but takes a somewhat different approach. HydroJ does not address the issue of interface evolution, focusing instead on supporting parameter evolution only; in contrast, Obje's focus is on rethinking how interface agreements are made as a whole, and how such agreements can more flexibly accommodate service evolution.

A number of systems designed to support *service composition*—the easy assembly of higher-level functionality by piecing together networked services—also touch on issues similar to those addressed by Obje. Omojokun and Dewan's [2003] framework for service composition provides mechanisms for interoperability that do not require agreement on service interfaces. Rather,

their framework allows "composers" to be created that rely on *programming patterns* assumed to be used by the services on the network. For example, if multiple services support operations with the same name, such as *powerOn()*, a composer can detect this and provide a unified mechanism for invoking this operation across multiple services. Other composers can detect and create composite interfaces for other patterns, such as data transfer across services, conditionally triggering service operations, and so forth. In essence, this framework shifts the burden of agreement from knowledge of the syntax and semantics of service interfaces, to knowledge of naming conventions for operations and parameter types. Thus, it can support the composition of services based on mechanical textual matching, and without requiring knowledge of the semantics of those services. Of course, developers must follow the prescribed naming conventions for this approach to work.

Ninja's "automatic path creation" [Gribble et al. 2001; Mao and Katz 2002] and the closely-related service composition work at Stanford [Kiciman et al. 2001] take a data-flow approach to service composition, and come the closest of any of the work we have described to our model of recombination. In particular, Kiciman et al.'s stated goal of "zero code" composition is strikingly similar to the goal of placing the user in the loop of deciding when and how to carry out interoperation. However, the data-flow model for service composition seems to pass significant complexity along to the users, requiring them to understand and explicitly compose pipelines of data transformations. These systems also take a different approach to dealing with potential protocol and data type mismatches, namely by introducing nodes in the service composition path that transcode from one protocol/type to another, rather than Obje's approach of using mobile code.

A number of systems have explored user interface approaches for dealing with device-rich environments. The work of Nichols et al. [2002] on the *personal universal controller* leverages a declarative abstract UI specification that can be targeted at runtime to different modalities. As another approach, Jini provides mechanisms that allow clients to acquire from a service mobile code that implements a custom UI for that service [Venners 2005]. Under the Jini model, each service on the network can provide a service-specific UI that can be used by any client on the network, as long as that client has knowledge of the programmatic interface implemented by the newly-acquired mobile code. Under both of these approaches, UIs are typically per-service, in that they are acquired from a single, specific service and do not provide composite user interfaces for controlling multiple services, or for controlling interactions among services.

Other approaches have been explored in the context of the iRoom. The iCrafter system [Ponnekanti et al. 2001] allows generation of multidevice interfaces based on "patterns" of service interfaces, allowing the system to create composite interfaces that unify interactions across multiple disparate services. While iCrafter shares similar goals with the work we describe here, namely the ability to provide on-demand user interfaces to clients that allow them to control multiple devices, the approaches are different. iCrafter relies on UI generators that can detect common patterns and emit usable unified

interfaces; our approach is to place more of the burden on the devices and services themselves, and rely on simpler composition mechanisms to unify these interfaces.

Others have explored multidevice interaction in the context of the web. Han et al.'s WebSplitter system [Han et al. 2000], for example, allows collaborative browsing of web pages across multiple devices. WebSplitter provides an infrastructure that allows web pages to be split and distributed across multiple devices that may belong to a single user, or multiple users, allowing interaction with multiple parts of the page from different devices. Fundamentally, however, this system is not about *integrating* control of multiple networked resources into a single UI, but about taking a single UI and *splitting* it across multiple devices. Other web-oriented systems include Bandelloni et al.'s [2005] work on automatic generation of migratory web interfaces. That approach is based on a proxy server, reminiscent of that of Fox et al.'s work [1998] on "transformational proxies" to "retarget" existing application UIs to new platforms, such as PDAs. Bandelloni et al.'s [2005] work builds on existing proxy-based approaches to support migration of the interface across new devices and to new modalities. Much like Bharat and Cardelli's [1995] work on migratory applications, this approach shares some surface-level similarities with Obje (support for device-rich environments), but fundamentally is aimed at addressing problems of migration rather than interoperation.

5. THE OBJE INFRASTRUCTURE

This section describes the basic design of Obje. We first examine the low-level capabilities that we expect to be built into devices and applications that implement the Obje software stack; then we describe how these low-level capabilities are wrapped in a programming model that supports the creation of dynamically extensible devices and applications. We note that the design target for the Obje platform is primarily network-connected devices (meaning devices with either a wired or wireless network interface and standard TCP/IP capabilities) with a modicum of processing power (typically meaning an embedded-class processor such as might be found on a set-top box, Internet appliance, or mobile phone). While we do not target extremely low-end devices (such as lightswitches, or simple sensors, or devices without network connectivity) directly, we have designed our architecture to accommodate such devices through a proxy mechanism, as described shortly.

The key distinguishing feature of our middleware platform is that it allows runtime extensibility of devices and applications, allowing new devices that enter the network to provide code to peers to allow them to interoperate with the new device.

This ability is provided through a *bootstrap protocol*, layered on top of TCP/IP, that provides a number of operations designed to support runtime extensibility. Most importantly, the protocol allows a new device on the network to provide a peer with the following:

—an implementation of a new application layer protocol needed to communicate with the device;

—an implementation of one or more *type-handling* modules, to render or process media or other data received from the new device;
—an implementation of new user interface controls, which can be used to control the device remotely; and
—a transparent bridging mechanism that can allow a peer to acquire new discovery protocols, or the ability to interact with devices that may not exist on the IP network, or that may not directly support the Obje software platform.

We call this our *bootstrap* protocol because it is used only for the initial negotiation and transfer of new capabilities necessary for compatible communication. Once this initial bootstrap transfer has completed, two devices communicate with each other directly using these new capabilities, and the bootstrap protocol is not used for further communication.

As noted earlier, these new capabilities are in the form of mobile code: self-contained executable content delivered over the network to the peer device. The Obje platform itself is agnostic to the format of this mobile code: the platform allows for a variety of code formats, including platform-independent code (e.g., Java bytecodes) as well as highly-tuned, platform-specific code (which of course would only be executable on a compatible target device).

Coupled with this bootstrapping mechanism for delivering new capabilities to peer devices, we have also developed a security framework for encryption and authentication without the need for centralized trusted third parties; this framework is intended to allow device creators or application developers to easily experiment with a range of application-layer access control policies that support the ad hoc, decentralized model of communication that is our goal.

Devices can participate in the Obje platform in one of two ways. First, to participate *natively*, devices must carry an implementation of the bootstrap protocol, may have one or more versions of mobile code intended for use by peers (these would typically be carried in some form of stable storage, such as firmware, flash, or on a disk), and may optionally have the ability to execute code received over the network. Second, and as we explain in later sections, our architecture also provides for non-Obje devices to participate in the platform *indirectly* through proxies provided by a host computer or other device.

5.1 The Obje Bootstrap Protocol and Code Formats

The Obje bootstrap protocol is defined as a profile on the Blocks Extensible Exchange Protocol (BEEP) [Rose 2001], which is a generic application protocol framework for bidirectional, connection-oriented communication. BEEP provides a number of facilities that Obje relies upon, such as message framing, asynchronous messaging, and a range of TLS-based security features that provide message integrity and privacy as well as authentication of peers on the network.

Obje devices advertise their presence over the local link via the widely-used Zeroconf discovery mechanism, which is based on multicast DNS (mDNS) and DNS Service Discovery (DNS-SD) [IETF 2005]. Device advertisements take the form of Uniform Resource Identifiers (URIs) that indicate the IP address and

port number of a BEEP endpoint on that device. Once a URI for a given device has been discovered, an Obje peer may communicate with it using the bootstrap protocol. This initial communication is started through a *FetchRequest* message to the peer, which in turn responds with its *ComponentDescriptor*. ComponentDescriptors are short XML documents that provide descriptive information about a device (name, icons, and so forth) as well as information about which roles the device may play (source or recipient of data, and so forth, as described below), and any mobile code that may be provided by the device.

We call these bundles of mobile code *granules*, and they are represented in the bootstrap protocol by elements called *GranuleDescriptors*. Each GranuleDescriptor indicates a location from which the mobile code may be loaded (typically, from the device itself, although this mechanism allows a device's code to be loaded from a third party on the network), as well as parameters used to initialize loaded code granules, a universally unique version identifier that may be used by clients to cache code granules, and a specification of the platform requirements of the code granules.

Most devices on the network will have the ability to send or receive data from other devices; devices that can do so declare in their ComponentDescriptors any content types that they may be able to process "natively," meaning, without the need to acquire any code granules from a peer in order to interpret the received data. These declarations are in the standard MIME format [Borenstein and Freed 1992]. This mechanism allows for the creation of devices that can participate natively in the Obje protocols, but do not require the ability to download and execute mobile code; this trade-off means that such Obje-compatible devices can be built more cheaply, but at the cost of losing the runtime extensibility that mobile code provides. For example, a small viewing device may declare that it can accept JPEG and PNG image data only, and refuse to accept (or be unable to process) granules that could extend its type-handling behavior to other image formats.

Depending on the roles a device plays, it may provide a number of types of granules to its peers to adapt their behavior in specialized ways. For example, a device that can act as an originator of data (called a DataSource) may be able to transmit specialized granules that provide peers with new protocol implementations or new type-handling behavior (including new CODECs) as described in Section 6.1, *Data Transfer*. Other sorts of devices may provide custom UI implementations or custom discovery protocols other than Zeroconf (see Sections 6.3 and 6.4, *Aggregation* and *User Control and Metadata*, respectively). Obje defines a fixed number of device roles, and thus a fixed number of granule types. Devices that play a given role are written to provide or accept the granule types defined by that role.

Table I shows an overview of the different device roles and the corresponding granule types used by those roles. Devices that can participate in data transfer implement one or both of the *DataSource* or *DataSink* interfaces; such devices support extensibility on support various aspects of data transfer, using *Session*, *Typehandler*, and *Controller* granules. Devices that can provide access to other devices, for example by encapsulating a new discovery protocol, support the

Table I. The Four Primary Modes of Extensibility Defined by Obje
(described as four "roles" in which devices can be used; devices that play one or more of these roles can provide or use a fixed set of granule types. Devices may participate in multiple roles)

Capability	Device Roles	Granule Types
Data transfer extensibility	DataSource DataSink	Session Typehandler Controller
Discovery protocol extensibility	Aggregate	ResultSet
UI extensibility	Component	UI
Metadata extensibility	Component	Context

Aggregate role, which provides extensibility in how peers acquire access to other devices on the network through *ResultSet* granules. Devices that provide access to custom user interfaces to control them, as well as to descriptive metadata, participate in the *Component* role, which is considered a base-level role that all devices should support; these devices use *UI* and *Context* granules to support extensibility along these dimensions.

Once a granule, such as a new protocol implementation, is transferred to a peer, it is executed directly by that peer. Thus, after the initial bootstrap phase to exchange any code necessary for compatibility, two Obje peers can communicate directly with one another, using whatever protocol- and data type-handling behavior is implemented by the granules provided by the source device.

Effectively, this mechanism allows peers to be built against a static protocol specification (the bootstrap protocol), which is then used to exchange new capabilities necessary for compatibility (in the form of granules) as new peers enter the network. It is this approach that allows devices to be coded against a fixed protocol, and fixed set of granule types, and yet be extensible to support new devices encountered "in the wild".

5.2 The Obje Programming Model and Runtime

We have developed a programming model and runtime software stack that wraps the low-level bootstrap protocol, along with other aspects of our infrastructure including remote code-loading and discovery extensibility. In this section we describe this programming model as well as a Java-based implementation of our runtime. Our programming model not only allows easier creation of applications, but also maps the capabilities of the platform into a polymorphic, object-oriented framework: devices on the network appear to applications as objects in their local address spaces; the roles those devices can play are mapped onto a fixed set of interfaces implemented by those objects; the methods in those interfaces use and return objects that themselves expose well-known interfaces, and which are implemented by the granules returned from devices on the network. Thus, to applications, the loading of code granules is transparent, appearing simply as new, polymorphic implementations of already-known interfaces.

This approach of transferring necessary implementations of known interfaces across the wire, rather than through standard single address-space method calls, is similar to Java's Remote Method Invocation (RMI) framework [Wollrath et al. 1996]. However, our implementation differs from RMI along a number of key dimensions. First, it is not specifically tied to the Java

programming language, and can support mobile code in a number of formats. Second, it neither provides nor requires the distributed garbage-collection facilities of RMI; code transferred to an application is used to create a new instance of an object in the application's address space, rather than a reference to a remote object that must be factored in to a reachability analysis for distributed garbage-collection. Third, it does not use serialization (which can often be fragile, especially in the face of object versioning or the need for multiplatform support) to transfer instance data across the wire; only implementations are moved.

Devices are represented in the Obje programming model by *components,* which are simply objects that reside in the address space of the client applications that use them. Components can be thought of as *proxies* for accessing a device such as a projector, printer, or PDA. This programming model is implemented by a small messaging kernel that forms the Obje runtime, and against which applications are linked. The messaging kernel implements the Object bootstrap protocol, and is responsible for creating new component proxy object representations within the client application's address space; this representation is created from information contained in the device's ComponentDescriptor. The kernel, upon receipt of the Component Descriptor from a device, generates a proxy object that represents the new device, and notifies the application that it is available via a simple event interface.

The component proxy objects that are generated by the kernel implement one or more programmatic interfaces that allow applications to access information about the remote device, as well as to acquire mobile code from it. Thus, while applications operate on these component objects using normal local method calls, these calls are translated into wire messages in the bootstrap protocol by the messaging kernel, and are sent to the backend service or device. For example, invoking one of the data transfer-related interfaces on a component (as described below) causes a request for the necessary granule to be encapsulated into the bootstrap protocol and sent to the remote device, which then returns the code to the client.

Figure 1 illustrates the process. Here, the messaging kernel in the application first discovers the device, and then acquires its Component Descriptor, which the kernel uses to create a new component object that acts as a proxy for the device. The application can interact with this new component object to query its name and other descriptive information, as well as to obtain mobile code from it that can be used to extend the application's behavior in certain prescribed ways.

When mobile code granules are transferred to a device, the messaging kernel exposes these as objects that implement a set of interfaces that define the ways in which clients can interact with new implementations that specialize their behaviors. These objects provided by the messaging kernel are simply "wrapper" objects that call into the granule, and implement the well-known interfaces defined by that particular granule type (as indicated by the rightmost column in Table I). Thus, applications written against the high-level programming model never see granules directly, but rather normal Java objects that implement well-known interfaces, but whose implementations come from granules delivered over the wire.

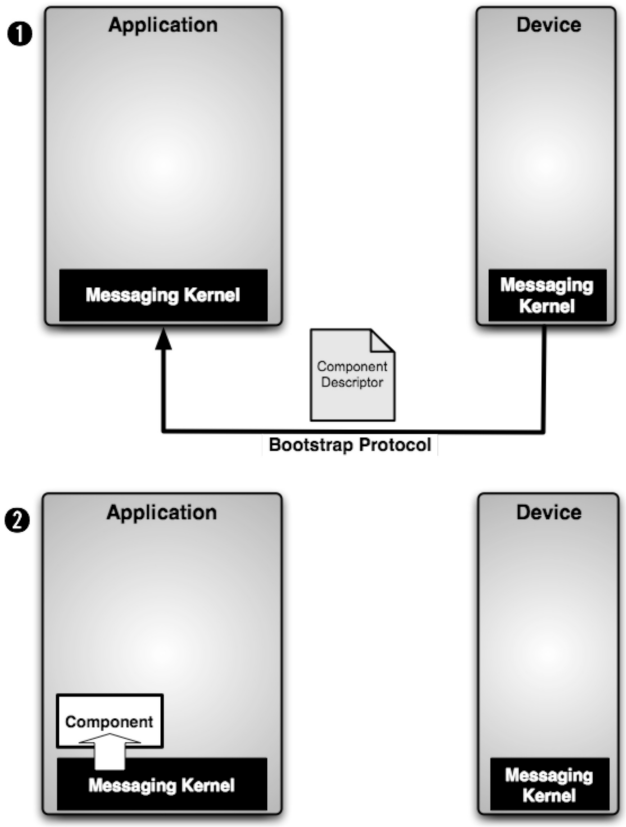

Fig. 1. The application on the left discovers the device on the right. A Component Descriptor, passed via the bootstrap protocol, encapsulates information describing the device (1). The Component Descriptor is used by the messaging kernel in the application to create a component representation as a proxy for the device (2).

Figure 2 illustrates the process of acquiring and using granules from the application's perspective. Here, application code interacts with a component object, which acts as a local proxy for the device shown on the right. Local method calls on the component cause the messaging kernel to request necessary granules from the back-end device, which are then returned to the application; both the request and the response are transmitted over the bootstrap protocol. Once the granules are returned, the messaging kernel creates a wrapper object that delegates local invocations of well-known methods to the code in the granule. In the case shown in Figure 2, granules provide a custom protocol implementation, custom data type-handling code, and a custom user interface for interacting with the device; in essence, these granules represent polymorphic implementations of the interfaces known to the client, transfered over the wire from the originating device. Once the new protocol granule has been loaded, the application communicates with the device using the protocol implemented by that granule, rather than the bootstrap protocol.

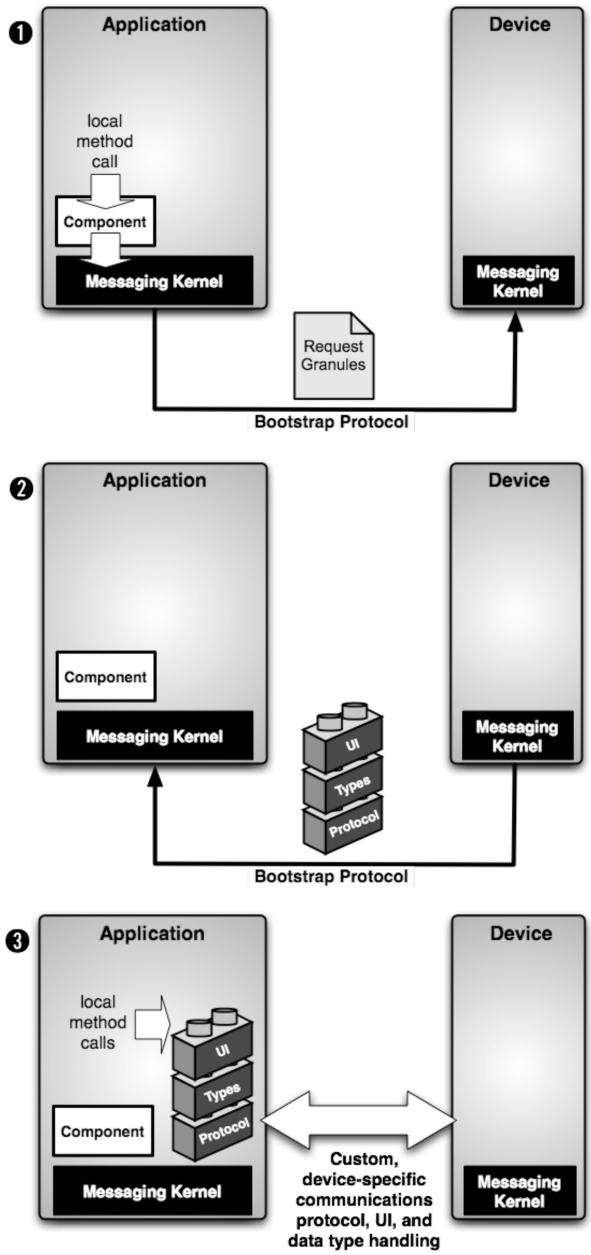

Fig. 2. The application interacts with the component to retrieve granules from the device, which allow the application to specialize its behavior for protocol, data type, and UI handling. Here, the application invokes a local method call on the the component that causes it to request a granule from the backend device (1). That device returns granules that provide new protocol implementation, data type-handling behavior, and UI specific to controlling communication (2). These are loaded into the application and returned as granules via the local method call on the component, where they can be used by the application (3).

In our current implementation, the messaging kernel is implemented in Java and produces component proxy objects that implement a small, fixed set of Java interfaces corresponding to the device roles described above. Thus, our high-level programming model most easily supports applications written in Java, although as noted we do support transfer and loading of native mobile code (described in Section 6, *System Experiences and Reflection*). Because the core protocols and wire formats of Obje are language-independent, devices and applications can be written against the bootstrap protocol directly in languages other than Java.

The next sections describe the patterns used by devices in specific roles to support extensibility of data transfer, discovery, user control, and metadata.

6. OBJE DEVICE ROLES AND COMMUNICATION PATTERNS

This section describes the patterns used by devices in specific roles to support extensibility of data transfer, discovery, user control, and metadata.

6.1 Data Transfer

The most important (and most complex) Obje mechanisms are those that support extensible *data transfer* between devices, such as a PDA sending data to a printer or a video camera sending a video stream to a fileserver. These mechanisms have been successfully used in a wide range of devices and client applications (see Section 7, *The User Experience of Recombinant Computing* for details), and provide runtime extensibility along three important dimensions: extensibility to new protocols, extensibility to new data types, and extensibility to new user interfaces for controlling a data transfer. The next three sections discuss each of these in turn.

6.1.1 *Protocol Extensibility.* Obje devices can play two roles in a data transfer: *data sources* and *data sinks*. These roles dictate how the devices will exchange mobile code during a data transfer: data sources provide new mobile code-based protocol implementations, which are then used by data sinks to retrieve data from the source using the new protocol. In the Obje terminology, this new protocol implementation is carried in a type of granule called a *session*.

Since many connections between devices are initiated as a result of some user action at a client, the basic communications pattern supports transfers started by a third party such as a remote control device, browser, or other application. In this pattern, a client requests a session granule from a source. The client will then pass this session to a data sink device to start the transfer. From this point, the source and sink exchange data directly, without it passing through the client. Figure 3 illustrates how a session is passed from source to sink by way of a client application that initiates the connection.

Note that the act of requesting a session from a source, as well as passing it to a sink, will involve a number of messages in the bootstrap protocol. Specifically, this sequence of operations will cause a request for the session granule to be sent to the remote source device, the mobile code for the session granule being returned over the network from the remote source to the client, and then passed

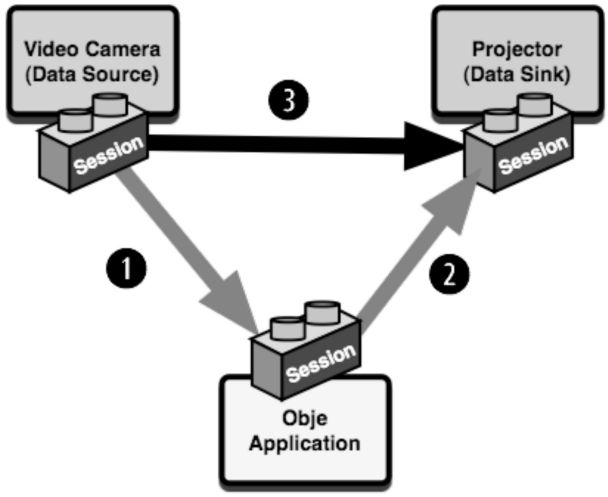

Fig. 3. Data transfer initiated by a client application. In step 1, the session granule is requested from the source by the client application; in step 2 it is provided to the receiving device. Both of these transfers happen over the bootstrap protocol. In step 3, the source and sink communicate directly using the protocol implemented by the source's session granule.

from the client to the remote sink. The Obje messaging kernel performs these requests as the client invokes operations on the component objects in its address space, hiding the details from the client.

In terms of our programming model, data source components provide a method that allow clients to list the formats of the data they provide (in the form of MIME types [Borenstein and Freed 1992]), as well as a method to return a new session granule from the device. Once a client has retrieved the session granule from the source device, it can pass it to any data sink device through a method defined on sink components. The session granule is then marshaled and passed over the network to the receiving device, which unmarshals it and invokes the code within it to read data from the original source.

Since the source device provides the session granule, it effectively has control over *both* endpoints of the communication, allowing it to use whatever protocol is appropriate for the type of data being transferred, with neither the destination nor the initiating client having to have *a priori* knowledge of that protocol.

In addition to providing protocol implementations, session granules also support a number of operations that allow clients to control the transfer of data between devices. Specifically, clients can terminate a session (stopping the flow of data), and can also subscribe to receive notifications about changes in the state of the transfer (that it has failed, for instance). In essence, the session acts as a capability, allowing any party that holds it to change its state, or be informed of changes in its state. Distribution of state updates happens in a semicentralized manner: updates cause a message to be sent to the source that created the session, which sends the update to other holders of the session. As described below, this same mechanism is also used to asynchronously distribute user interface granules to devices involved in the transfer.

6.1.2 *Data Type Extensibility.* The features of the data transfer pattern outlined above (independence from specific protocols, the ability for third parties to initiate connections, and the ability to receive notifications about changes in the state of a transfer) are necessary but not sufficient for providing the flexible and seamless version of recombination that we envision. Namely, it allows easy, protocol-independent interconnections among components, but only insofar as those components understand the same data types.

Just as we believe that future devices will bring with them new protocols, an ability to work with new data formats and media types is also required. If components must be prebuilt to understand each other's data types in order to work together, we drastically limit their ability to interoperate in a truly ad hoc fashion and to evolve to support arbitrary new devices. We must move away from the requirement that devices must be replaced or manually updated each time a new data format appears on the network.

There are a number of approaches one might take to overcome this conundrum of extensibly handling new data types. One approach (reminiscent of Ninja's paths [Gribble et al. 2001; Mao and Katz 2002]) would be to allow *filter* services such as Ockerbloom [1998]) to exist on the network that accept data in one format and translate it to another. We intentionally avoided this approach, however, first because it requires the presence of additional infrastructure on the network (the filter services themselves), but primarily because of its implications for the user experience. We wanted to avoid a data-flow model that required users to explicitly connect through a chain of format conversion filters. Our initial user studies led us to believe that most nontechnical (and even many technical) users did not easily grasp such a model; on the other hand, automated filtering would be problematic because our desire to allow semantic extensibility would require users to be involved in selecting among multiple (potentially semantically incompatible) filters.

Instead, the approach taken by Obje is to use mobile code to allow for extensibility to handle arbitrary data types, without the need for excessive user involvement, and especially without the need for a "wiring diagram" style of connection. Obje allows devices to broaden their statements of compatibility beyond simple MIME types. For example, a projector—a device that by its semantics is designed to display things—might claim that it can understand not only JPEG data, but also that it understands the semantics of other things that are displayable. In other words, a device might claim that it understands some abstract representation of a set of operations that it can perform on data without having to understand the data itself.

These widened statements of compatibility are declared as programmatic interfaces in the list of understood types. For example, a projector may list not only that it can accept JPEG data (MIME type *image/jpeg*), but also that it is written to use granules that implement a Viewer interface that it will invoke to display data (which we likewise express as a custom MIME type, *application/x-obje-typehandler-granule; representationClass=com.parc.obje.datatransfer.Viewer*). By declaring that it understands a particular interface, a sink indicates that it is written to understand and use objects that implement that interface. Likewise, a source that declares that it can provide a particular interface means

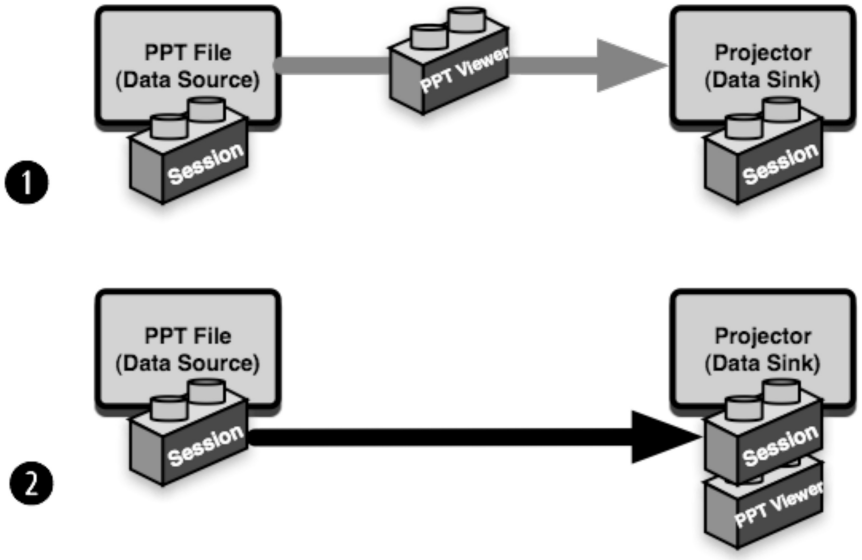

Fig. 4. A projector device uses a typehandler to process data in an unknown format. Here, the session granule has already been transferred to the sink device. When data is delivered in Powerpoint format, a typehandler granule implementing a Powerpoint Viewer is transferred over the bootstrap protocol (step 1). After this, data is transferred between source and sink using the protocol implementation provided by the session granule, and the received Powerpoint data is rendered via the typehandler Powerpoint Viewer granule (step 2).

that it can transfer a granule that provides a specific implementation of that interface.

We call these mobile code-based implementations *typehandlers*, because they provide code that allows a receiver to process a previously unknown data type. In essence, this extension of the type system allows sources and sinks to negotiate richer interfaces to the data they exchange, and allows data type-specific implementations of these richer interfaces to be acquired at runtime. If both the source and sink agree on an interface they understand, a mobile code-based implementation of this interface will be transferred from the source to sink as a granule, and invoked by the sink to handle the data.

The set of interfaces that these typehandlers may implement is open-ended and extensible. This is the same as with MIME types: there is an extensible set of them, new ones will come over time, and they must be known to the involved parties for communication to occur, but no one else need understand them.

Figure 4 shows an example, displaying Powerpoint format data on a projector that is not explicitly written to use such data. In this case, the projector is written to understand a number of raw data types (GIF and JPEG, in our current implementation), and is also written to understand objects that implement an interface called Viewer. Any party that can provide a Viewer wrapper around its data can thus connect to the projector. Here, the projector downloads and a specific typehandler granule that, once instantiated, provides an

implementation of this interface that renders Powerpoint, a format previously unknown to the projector.

The use of typehandler granules doesn't solve all problems with type compatibility—in the example above, even though the projector doesn't have to understand Powerpoint directly, it must still be written to understand the Viewer interface. The typehandler approach does, however, provide a number of concrete benefits over simply requiring agreement on simple data types. Most importantly, typehandlers provide a means for dynamic extension of the set of types that can be used between two devices. As long as a sink is written to accept a typehandler interface, components that were previously incompatible with it can be made compatible through the addition of a typehandler that meets that interface. Such easy, dynamic extension of type compatibility is not possible when only static types are allowed, without rewriting either source, sink, or both.[3]

Such dynamic extensibility is especially important when a source provides an unusual format. For example, one of our Obje services provides a live video stream of a computer's display, using VNC [Richardson et al. 1998]. We neither require nor expect all receivers to be able to parse and process VNC data, so the source provides a typehandler that implements a Viewer interface. Through this interface, any sink that is written to understand the semantics of Viewers can accept and display a live VNC stream.

6.1.3 *Transfer Control Extensibility.* The final aspect of extensible data transfer behavior concerns how we can *control* arbitrary aspects of a data transfer beyond simply starting and stopping the transfer. For example, consider a generic browser-style application created using Obje. If a user uses this application to connect a file of Powerpoint slides to a projector, we would like to display to the user controls *specific to that interaction*. This may include UIs for the particular model of projector as well as for the slide show. Of course, we need to be able to do this without requiring that the browser be specifically written to understand the details of projectors, or of Powerpoint slides. This ability to acquire and display arbitrary per-device user interfaces is necessary, given the user-in-the-loop philosophy that is fundamental to the Obje approach.

Obje uses the same state notification mechanism defined by session granules to deliver yet another type of granule, which we call *controllers*, to any of the parties that hold a copy of the session object [Newman et al. 2002a]. Using this mechanism, any party that holds the session can "add" a controller granule, which causes it to be delivered to any other parties holding the session that have solicited an interest in receiving such granules; this arrangement allows user interface code to be delivered asynchronously over the network, at the time it is needed and for presentation by whichever client is managing user interaction.

[3]A secondary potential benefit, although one we have not explored in depth, is that by "wrapping" data formats in programmatic types, there is the opportunity for more fine-grained machine reasoning about type compatibility, using methods such as those proposed by Wing and Ockerbloom [2000] based on formal notions of subtyping and type compatibility [Liskov and Wing 1994].

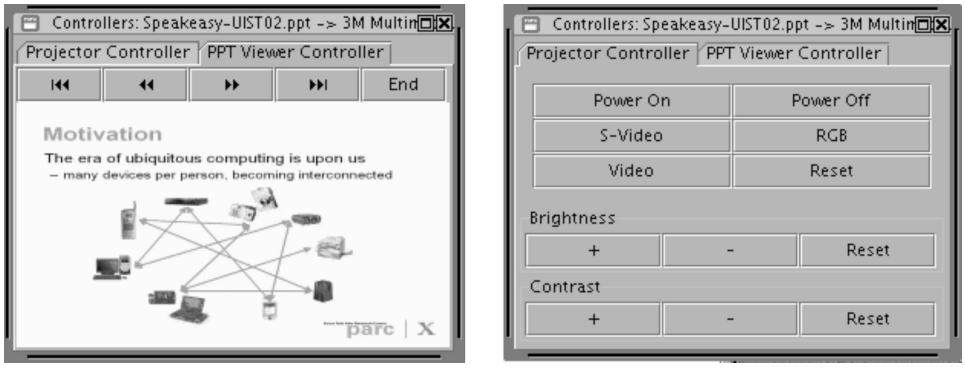

Fig. 5. Projector and slide show controllers running on an iPaq PDA. The client application solicits to receive controllers from any session in which it is involved. Received controllers for a given session are presented in tabbed panels.

A key advantage of this approach is that controllers can be added by any party that holds the session, in the same way that any party that holds the session can also update its state. This means that sinks, sources, and typehandlers can all add controllers. For example, in our Powerpoint case, when a connection is established to the projector, the projector (sink) component may add controls for adjusting brightness and other projector parameters. The typehandler—which is the only party in this scenario that would understand Powerpoint data—would add the slideshow controls. Both would be transmitted over the network to the application that initiated the connection.

Figure 5 shows a set of simple controllers from the Powerpoint-to-projector example, running on a PDA. Note that the controller delivered to the PDA can provide potentially arbitrary functionality, based on the code that is delivered to it. In this case, the Powerpoint controller displays miniature versions of the current slide and notes on the PDA screen, even though the PDA is not coded to understand Powerpoint files, and does not even have the application installed. This controller also allows the user to draw on the slide using a stylus; the strokes are communicated back to the Powerpoint typehandler executing on the projector, where they are rendered over the slide.

Together, the dimensions of runtime extensibility provided by the Obje data transfer mechanisms (extensibility to new protocols, new data type-handling behavior, and new UIs) allow applications and devices to richly interact with each other, while requiring only minimal *a priori* knowledge of each other.

6.2 Aggregation

The second major group of mechanisms in Obje supports *aggregation*. Aggregates are components that appear as logical collections of other Obje components. In Obje this pattern is used in a wide range of situations: to access filesystems, which appear as collections of components representing files and folders; to support devices that encapsulate new discovery protocols; and also to support devices that provide access to nonIP networks and legacy (nonObje) devices. Any situation in which a device provides access to other devices uses this interface.

From the perspective of our programming model, applications initiate an interaction with an aggregate component by performing a *query()* method call, defined by the Aggregate component interface. Applications pass a parameter that allows them to match devices based on their type, or metadata associated with them (see Section 6.3, *User Control and Metadata* for details on device metadata), allowing applications to only be notified of a subset of available devices that pass some filter. The query operation returns a *ResultSet* to the requesting application. ResultSets are granules that present a simple dictionary view (component IDs as keys and components as values) of the devices that match the query. Once an application has a ResultSet, it can iterate through the "contained" devices and solicit notifications about changes in the set of devices that the original query matches, allowing push-based notification of new devices. Thus, the semantics of ResultSets are that they are "live," and may be continually updated as matching devices come and go.

In our high-level programming model, initial Zeroconf-based discovery is presented to applications as a "root" Aggregate component that contains all Zeroconf devices on the local link, and supports the same query and notification mechanisms described here. This root aggregate is instantiated and provided to applications by the Obje runtime stack to bootstrap their discovery of devices on the network.

Because ResultSets are implemented as mobile code-based granules, this simple pattern can support a great deal of extensibility and power. For example, this mechanism allows applications to take advantage of arbitrary new discovery protocols and to interact with devices on physical networks that the applications themselves do not have direct access to. The sections below describe how these scenarios work in Obje.

6.2.1 *Discovery Extensibility.* While Obje devices support Zeroconf as their standard discovery mechanism, such a "one size fits all" approach is untenable in a world with a rich variety of devices and networked environments [Edwards 2006]. As a simple example, Zeroconf does not provide the ability to easily discover devices outside the local link; in such cases, it may be useful to support discovery mechanisms that use a registry service, which can allow better adminstrative configuration over which devices are discoverable.

Thus, one key application of the aggregation mechanism is to provide alternative means of discovery for Obje devices and applications. This model allows a device or service to be deployed onto the network; once discovered normally via Zeroconf, this device or service can provide to its peers new mechanisms for discovery via ResultSet granules. In effect, existing clients on the network gain the power of new discovery protocols automatically via deployment of a single device or service, and without requiring that they be aware of the new discovery mechanisms.

In the most simple case, the ResultSet granule delivered to devices can simply forward the operations performed on it to a remote device or service that performs discovery on its behalf. This allows a device or service to serve as a discovery bridge—it can perform discovery using *arbitrary* other protocols and return the results to unmodified applications in the form of standard Obje

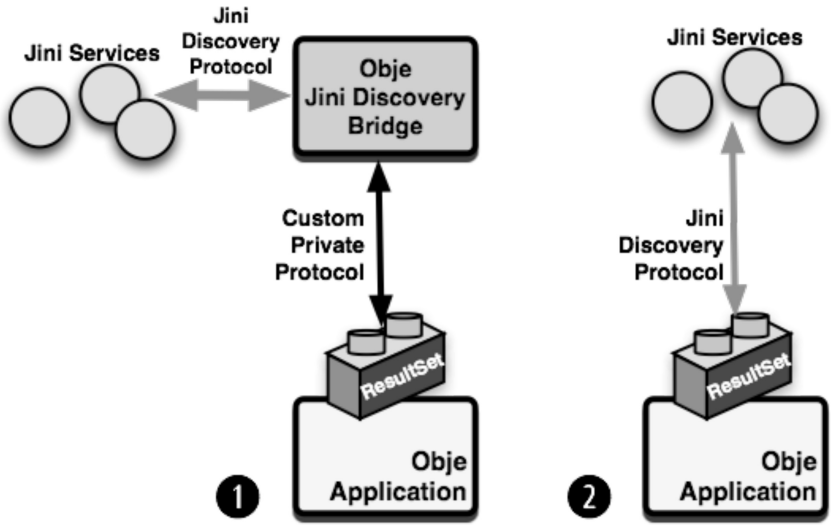

Fig. 6. Two common discovery patterns using ResultSets. On the left, a ResultSet granule acts as a simple "shim," providing access to a separate discovery service running on a remote machine; the ResultSet implements a private protocol for communicating with the remote service. On the right, a ResultSet granule implements a custom discovery protocol that executes directly in the application.

ComponentDescriptors. In other cases, the ResultSet granule may itself provide a custom implementation of a new discovery protocol that executes directly in the client, thereby allowing clients to *directly* discover other peers that may have been inaccessible to them previously.

Figure 6 illustrates both of these cases. In the first case in Figure 6, a lightweight "shim" ResultSet communicates using a custom, private protocol to a back-end service that serves as a discovery bridge for Jini services; this back-end service invokes the Jini discovery protocol [Sun Microsystems 1999] and returns discovered services to the client. Because ResultSets leverage mobile code, however, other configurations are possible. For example, a ResultSet granule can provide a *full implementation* of a new discovery protocol, which can then be delivered to the client where it executes locally. In the second case in Figure 6, the discovery process does not happen in the external service; instead, it happens within the client itself, which has been dynamically extended to use the Jini discovery protocol through the code contained within the granule.

Whether discovery happens in the application or in some external service or device is up to the entity that provides the ResultSet granule. In either case, the application simply operates on the ResultSet using the standard iteration operations defined on it, and need not care how the custom implementation does its work.

6.2.2 *Legacy Device Support.* The Aggregate pattern is also used by Obje to provide access to legacy devices, meaning both non-Obje devices and devices

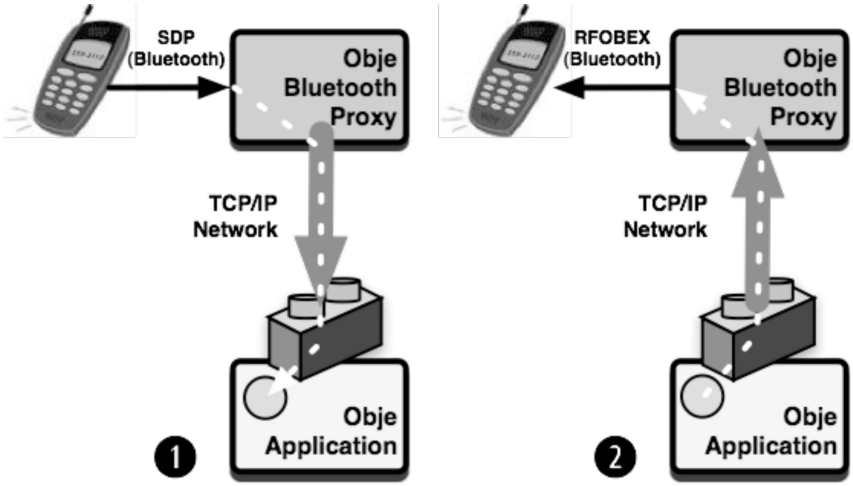

Fig. 7. Using aggregates to provide access to legacy resources. Here, a Bluetooth proxying service executes on a remote machine to provide access to legacy Bluetooth devices that do not run Obje. As the service discovers a Bluetooth device (on the left), a ComponentDescriptor is generated for it and delivered to the client application via the proxying service's ResultSet granule, embedded in the client. The client can operate on this proxy component just as it can any other component. On the right, data transfer operations invoked on the component are forwarded back to the bridge service, which uses the Bluetooth device's native protocols (RF-OBEX) to transfer data to it.

on non-IP networks. In these cases the Aggregate device acts as a proxy for one or more devices that do not themselves implement the Obje bootstrap protocol. This can allow, for example, USB connected devices on a PC to be exposed as if they were native Obje devices through a "USB Aggregate" that runs on the PC, understands how to communicate with these devices, generates ComponentDescriptors for them, and can participate in the Obje bootstrap protocol on their behalf.

Figure 7 illustrates a Bluetooth Aggregate. This is a service running on a machine with both a Bluetooth and a traditional wired network interface, which exposes itself as an Obje aggregate device. The ResultSet granule delivered to clients by this component will communicate with the back-end machine, where the Bluetooth discovery protocol (SDP) [Bluetooth Consortium 2001] is used to discover devices within range. As devices are discovered, the service generates a ComponentDescriptor for each of the devices, returning them to the client through the ResultSet, as shown in the first illustration in Figure 7. To the client, these aggregate-generated components are indistinguishable from any other component; the Bluetooth aggregate itself simply appears as a collection of all of the Bluetooth devices on the network. Operations on these proxy components, however, are relayed over the wired network to the remote machine, which then uses the Bluetooth Object Exchange protocol (RFOBEX) to communicate with the device. In this way, aggregates can play a proxying role, acting as an intermediary in interactions between the proxy component for a device and the device itself. This arrangement can allow the interconnection and use of arbitrary legacy devices, even on different networks.

We have created a number of such proxying aggregates for legacy devices, allowing full networked access to devices such as USB cameras, web cams, and music players, and Bluetooth phones and PDAs.

From the client's perspective, all of these varied uses are possible using the same aggregate interface, and without rewriting. At startup, Obje applications acquire a reference to a single "root" aggregate, provided by the Obje runtime, which allows access to an initial layer of devices discovered on the local link via Zeroconf. Some of these devices may themselves act as aggregates, providing access to collections of components made available via new discovery mechanisms, or via proxying legacy devices, potentially on nonIP networks. New discovery protocols can be made available (either directly, or indirectly through bridging) to all clients simply through a single new device on the network. Likewise, proxies for devices residing on different networks, or for devices that do not communicate using Obje at all, can be achieved through the addition of the necessary proxy aggregate to the network. We believe that this range of uses demonstrates the potential of combining fixed interfaces with mobile code—by installing one device on the network, all existing devices and clients can benefit.

6.3 User Control and Metadata

The data transfer and aggregation mechanisms are the primary way that Obje supports extensible and adaptable interdevice communication. However, Obje also supports two mechanisms designed to allow human users to more easily control and understand the Obje devices available on the network. Both of these capabilities are supported by the base *Component* role, implemented by all Obje devices at a minimum in addition to whatever other roles they may support.

The first of these capabilities allows client applications to directly request a UI from a device. The ability of devices to provide UIs for controlling them is a kind of "escape hatch," allowing Obje devices to provide access to functionality that cannot be easily represented using the data transfer or aggregation interfaces. Applications agree on the mechanisms for acquiring and displaying such UIs, but have no knowledge of the particular controls provided by any UI. This device-level UI mechanism is different from the UIs provided by controllers, which are specific to a single, ongoing communication between peers; here, the UI is meant to be a top-level administrative interface for the device as a whole. In the case of a printer for example, an application could request the printer's UI granule and invoke it to display a control panel to the user, allowing control over defaults such as duplex, stapling, and so on.

Since different client platforms support a wide range of possible UI mechanisms, Obje allows devices to provide multiple UI granules; applications select from these by specifying the platform of the desired UI. For example, a client running on a laptop might request a "javax.swing" GUI, while a PDA might request a "thinlet" XML-based UIs. The strategy is flexible in that it allows devices to present arbitrary controls to users, and allows multiple UIs, perhaps specialized for different platforms, for a given device. The primary drawback is that it requires device developers to create separate UI granules for each

type of presentation platform. A solution would be to use a device-independent UI representation, such as those proposed by Hodes and Katz [1999] or UIML [Harmonia Inc. 2000], and then construct a client-specific instantiation of that UI at runtime. We are not currently focusing on developing such representations ourselves, but rather on the infrastructure that would be used to deliver them to clients.

Obje also provides a mechanism that allows devices to provide arbitrary descriptive metadata about themselves. Since our premises dictate that the semantic decisions about when and whether to use a component must ultimately lie with the user, we must provide mechanisms to allow users to understand and make sense of the services available on the network. For example, simply knowing that a device can be a sender or receiver of data provides little utility if no other information can be gleaned about it. For this reason, Obje devices support the ability for applications to retrieve a granule from a device that provides *metadata* about that device, which may include such details as name, location, administrative domain, status, owner, and so on.

Our representations for metadata are very simple: metadata granules provide access to a simple map of key-value pairs, with keys indicating the names of attributes ("Name," "Location," and so on), and values that are arbitrary objects. The set of keys is extensible, as we do not believe any fixed set is likely to support the needs of all applications or components. Likewise, we neither require nor expect that all applications will know the meaning of all keys, nor share a common metadata ontology. The goal of this mechanism is primarily to allow sense-making by users, and only secondarily to allow programs to reason about metadata.

6.4 Security Framework

Obje's approach to interoperation raises a number of issues from the systems perspective, particularly concerning security. These range from the need to determine the safety of mobile code to concerns such as authentication and access control in peer-to-peer settings. For the issue of safe execution of mobile code, we consider ourselves to be consumers of solutions rather than providers. For Java-based code granules, we currently support the mechanisms provided by Java's security primitives. These are, however, fairly brittle in an environment of ad hoc interoperation. For example, the Java security system allows downloaded code to be granted rights based on where it came from, who is running it, or who signed it [Gong 1999]. Of these, the assignment of rights based on signing is potentially the most useful, although it requires shared notions of identity known to both the sender and receiver of the code (a common certificate authority, for example). The Java concept of granting access based on where downloaded code came from is largely unworkable under Obje, since Java uses a URI to indicate "where code came from". In a peer-to-peer environment, without stable IP addresses, much less verifiable domain names to identify peers, this model is lacking.

Sandboxing of code, a mechanism which is also supported by Java through the use of *security policies*, is likewise somewhat brittle under assumptions such

as ours. Different granules may require radically different resources from their hosts, and thus it is difficult to define a single security policy that will support the desired behavior—and enforce the desired restrictions—*a priori*. Perhaps even worse from our perspective is that users must necessarily be "in the loop" of device interactions; previous research (including Whitten and Tygar [1999] as a canonical example) has shown that the finer points of security are not well-understood by users. We do not believe it is tenable to require that end-users make decisions about the correct permissions necessary for execution of a given piece of code since, obviously, many users will instead disable security altogether, defeating any protections that might otherwise be in place.

Thus, while the common Java security features are available in Obje (at least for situations in which granules contain Java bytecodes), these features are insufficient for our model of interoperation; thus, we have focused on providing an alternative set of mechanisms that are more amenable to our vision of peer-to-peer connectivity and ad hoc interoperability. Specifically, our model follows three basic principles. First, to ensure data integrity and privacy, all communication among Obje peers must be both encrypted and authenticated. Second, to enable the fullest range of peer-to-peer applications, we need to be able to support authenticated and access-controlled interactions among devices that may have no *a priori* trust relationships with each other. For example, we do not require that they each possess certificates issued and signed by some common, centralized certificate authority (although this ability is available as an option if device manufacturers or service builders choose to obtain such certificates). Finally, it is the device offering its services that makes the final decision about which peers are allowed to access it; this determination must be made on the device itself.

The core approach we have taken is to create a general security framework as a part of the Obje core, which addresses these three principles. This framework provides mechanisms for encryption and authentication without the need for centralized trusted third parties, and is intended to allow device creators and application developers to easily experiment with a range of application-layer access control policies.

At the base level, Obje uses TLS to secure all interactions among components and to exchange and verify credentials that are used to enable access control decisions to be made in a distributed context. We use TLS in "client authentication" mode, which means that peers in an interaction will mutually authenticate each other. Obje supports a flexible model of identity for purposes of authentication: the platform can support a "device certificate" (meaning a certificate issued and signed by a known certificate authority) if device makers choose to do so. If such device certificates are not present, devices will generate cryptographic public/private key pairs along with a self-signed certificate that will be used to identify the device and secure future communication with peers.

Layered on top of this model, device and service creators can implement pluggable access control policies that allow the owners (or creators) of Obje devices or services to dictate who or what has access to these resources. To create a new security policy, developers must implement a small number of security-related

callback functions. These callbacks are invoked on a device when a peer attempts to access its functionality (such as starting a connection to or from it, requesting the contents of an aggregate, and so forth), allowing the device to make a local determination about who or what will access it. Common policies, for instance, may allow trusted access to all devices that have certificates issued by the same entity (such as a home certificate authority), or allow communication only with "whitelisted" peers, or restrict access to devices that have been indicated as "trustworthy" (for at least a limited set of operations) by a trusted third party.

In addition to such "standard" policies, to date we have used this framework to explore two novel end user-oriented security policies for pervasive environments. The first, used by the Casca tool [Edwards et al. 2002b], allows device owners to grant access to their devices to others based on a grouping model. Users create new collections or "groups" of devices that will be accessible to a set of users who are associated with that group. Each group has its own root certificate that identifies it and, when signed, can be used by devices and users to prove their membership in that group. These group credentials are used to secure communication between peers that are members of the group. The underlying security framework allows application developers to add such application-specific logic (the notion of groups, and group members, along with other features such as delegation) by creating a set of callback functions that work in concert with each other to implement the desired policy.

A second, more exploratory model, described in depth in Smetters et al. [2006], uses a device called the *Instant Matchmaker* to create secure relationships among peers; this system is specifically intended to allow secure connections in environments that may include both trusted and untrusted devices and sensitive and nonsensitive content. The matchmaker device acts as a trusted intermediary that can provide the necessary credentials to establish access rights to Obje devices. The device exploits proximity-based authentication [Stajano and Anderson 1999] to allow users to physically point to peers that should be allowed access to a given device; such a pointing gesture conveys the user's intent, which in turn causes the exchange of the required device credentials necessary to allow the user's desired action. This model supports *implicit security* [Smetters and Grinter 2002], in which there is no explicit, *a priori* determination of which principals have access to a given resource. Similar proximity-based interactions have been explored in other domains, where they have been shown useful as an approach to end-user security [Balfanz et al. 2002; Rekimoto et al. 2003; Swindells et al. 2002].

A final aspect of security that we have explored is the use of encrypted, and optionally signed, granules. These mechanisms can support a range of features. For example, encrypted granules can be used to allow interoperability among only a subset of devices. While such a technique may seem antithetical to our goal of open, ad hoc interoperability, such features are important in certain contexts such as consumer electronics, where device vendors may want to allow certain features to be supported only when used with devices also sold by them. Granule signing, while currently supported by Obje, has not been explored or exploited in depth. In theory, such code-signing facilities could be used to provide

an additional layer of security or robustness to allow devices to be configured to only accept code from peers that can be verified in some way.

7. THE USER EXPERIENCE OF RECOMBINANT COMPUTING

As noted earlier in this article, ad hoc interoperability comes at a cost: while it holds the promise of allowing interactions among devices with only limited beforehand knowledge of each other, it removes the current tight semantic integration that exists when devices are able to interact.

Thus, one of the key questions of this work is how usable is a world in which users assume the burden of determining when and whether devices should work with one another. We have explored this question through a combination of application building, deployment, and experimentation, pursuing the style of iterative development of infrastructure to reflect human-centered needs advocated in Edwards et al. [2003].

7.1 From Device-Oriented Interaction to Task-Oriented Interaction

Some of the earliest applications our team built were generic *browser* applications, which provide direct access to the Obje devices on the network. These applications present simple lists of discovered devices and map user operations in the UI onto Obje device operations: for example, dragging and dropping a source device onto a sink initiates a data transfer and displays any received controller UIs in the browser; double clicking an aggregate device "opens" it, revealing the components to which it provides access. We believe that such generic applications, which allow users to discover and interact with arbitrary devices in an ad hoc fashion, will play an important role in any ubiquitous computing future. In the absence of specialized applications for every conceivable task, a more generic tool—one without specialized domain knowledge—will necessarily play a part.

We have built and studied a number of these applications. The first (reported in Newman et al. [2002b] and Edwards et al. [2004] and shown in Figure 8), was a web-based application designed for access on a PDA. Our early experiences with this tool informed a number of our architectural design choices, particularly in our data transfer design. For example, early versions of Obje relied on external filter services, which could be chained when performing a data transfer. The UI designs that were implied by this mechanism required the user to "hand assemble" chains of data transformations to connect two devices in situations where their data types would have otherwise been incompatible. While other sorts of user interfaces could have been created, for example by automatically finding paths of filters, such arrangements remove the user from the process of determining *which* filter services to use, which is important in the situation where filters execute on the network on hosts with different speeds or different levels of trustworthiness. The user, not the system, is better equipped to consider these factors.

Feedback from users on our prototype UIs based on filter chaining led to the creation of the typehandler mechanism, which has the benefit that in cases where compatible typehandlers exist, initiating a data transfer requires only

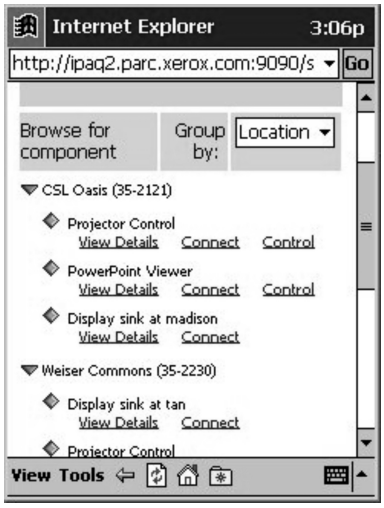

Fig. 8. An early Obje browser running in a web browser. Basic facilities are shown for sorting and selecting components by location.

the specification of the source and sink devices. This architecture essentially bundles the filter service code into a granule, which then executes in the receiving sink.

However, interfaces that directly expose such low-level operations are still difficult for users to work with. Our experiences showed that the basic concept of exposing data-flows between devices as a first class operation in the interface was confusing to many users. During our tests, for example, some users wanted to simply "open" a Powerpoint file once it had been located, expecting it to appear on the display in the room. These desires seem to betray expectations based on PC use, in which the tight coupling (both semantically and physically) between the PC and its display device allows actions such as opening a file to have a relatively unambiguous meaning. In a more loosely-coupled world, potential ambiguities exist, which must be resolved by the user.

These experiences seem to point toward a fundamental problem with creating usable interfaces for a world of ad hoc interoperability. Generic tools (i.e., meaning, tools that can work with arbitrary devices) are necessary to take full advantage of the infrastructure. And yet, by their nature, such generic tools do not have the tight semantic integration—the "understanding" of what certain devices or content types "mean" built into their programming—that supports good ease of use. Further, it requires that the underlying mechanisms provided by the infrastructure be understandable and usable by users if they are to take advantage of this power.

We believe that one approach to resolving this dilemma is to create facilities for moving from *device-oriented* interactions to *task-oriented* ones. That is, rather than specifying the various data flows among individual devices, the user would simply select a desired task and the system would instantiate the necessary device-to-device interactions. Of course, actionable representations of such tasks must exist for this option to be available. There are a number of ways such

Fig. 9. The OSCAR user interface. From left to right: the device and media list, device and media detail, connection list, connection detail, setup list, and setup detail screens.

representations could come into being. They could be created by experienced users much in the way that Excel macros for a given site are often created by local "gurus," and then shared throughout that site [Mackay 1990, 1991]. We have explored this approach, which we call *task-oriented templates*. These templates are XML documents that contain "slots" that describe the devices that are needed to accomplish a given task, in terms of input and output types, metadata properties, and so forth. When a template is instantiated, the template engine attempts to match available devices to slot descriptions (perhaps with help from the user), and then creates the necessary connections among them. For example, a "give a presentation" template would specify slots for projectors, speakers, and a source slides file; instantiating this template would create the connections among Obje devices to allow easy setup of a conference room.

One application we have developed, called *OSCAR* (for Obje Service Composer and Recomposer) uses such templates as the foremost feature in its interface [Newman et al. 2008]. In OSCAR, templates are presented to users as "setups" that they can instantiate to control and interact with networked media appliances around their homes. Figure 9 shows an overview of the OSCAR interface.

Of course, this approach still requires that someone have the knowledge, skill, and time to create templates for each common task. Another approach, which we plan to explore in the future, is to observe the connections users establish among devices and attempt to infer useful relationships from this observation. For example, if a user commonly creates a particular set of connections among devices, the system may store a generalized representation of this configuration, allowing the user to recreate it later. In essence, under this approach, templates are created by the system through a learning technique. The system mines the semantic information provided by users through their actions to create reusable descriptions of tasks.

More generally, we believe that the sorts of open-ended device use available through systems that support ad hoc interoperability point to the importance

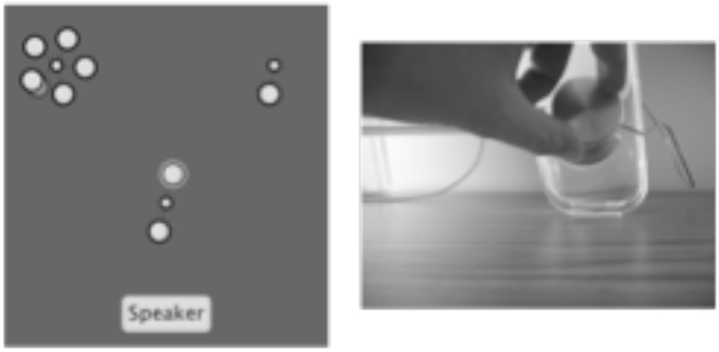

Fig. 10. The Orbital Browser's interface, and its input device. Spinning and pressing the knob allows users to navigate to any discoverable device, initiate and terminate connections, and so forth.

of *service composition* techniques. Service composition is the process of combining multiple, discrete services or devices together to achieve some higher-level goal. A number of other researchers have explored service composition in the context of web service composition [Chakraborty and Joshi 2001] as well as ubiquitous computing [Ponnekanti et al. 2001; Humble et al. 2003; Omojokun and Dewan 2003]. For example, the CAMP system [Truong et al. 2004] uses a novel magnetic poetry interface, allowing users to select and arrange available verbs and nouns to indicate desired compositions of services. We have also explored the design space of service composition interfaces through a number of tools that embody different interaction approaches to composition. For example, the Orbital Browser [Ducheneaut et al. 2006] (Figure 10) provides a lightweight interface that uses only a knob as an input device to select and compose services. This tool was intended to demonstrate a minimalistic interface for providing open-ended service composition. We argue that providing powerful, end user-oriented tools for service composition will become even more essential as we move to a world in which device connectivity is not limited by what the developers of those devices foresaw, but by users' desires and needs.

7.2 Specialized Application Domains

A second context in which we have explored the user experience of ad hoc interoperability is through applications intended for specific domains. Although we believe that generic tools will necessarily have an important role to play in future ubicomp environments, we also believe that there will always be a need for specialized tools that embody certain notions about how they will be used. While still open-ended in the sense that they can make use of arbitrary new devices that appear on the network, they frame users' interactions with these devices within a certain context.

For example, we have created an Obje-enabled set-top box for media-oriented applications [Edwards et al. 2005]. This system allows users to interconnect audio- and video-related services hosted on the box, as well as Obje devices and

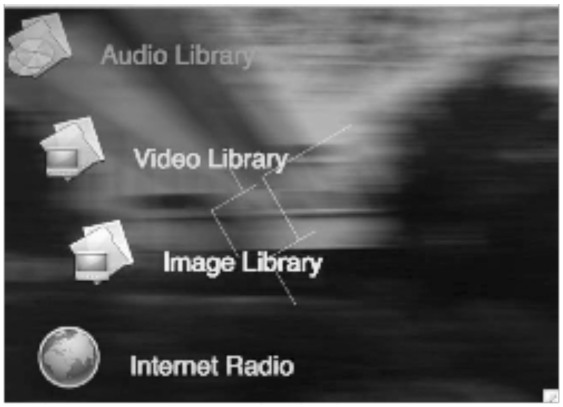

Fig. 11. Top-level on-screen user interface for our set-top box, showing top-level options grouped by media type. Users interact with the interface through a standard infrared remote control.

legacy devices elsewhere on the home network. By making assumptions about the context of use (storing and playback of media files), the interface can be streamlined somewhat: the system discovers and groups all devices and content available on the home network into either "audio" or "video" classes (depending on the MIME types they support), and organizes these into menus. Selecting a media source allows the user to either play the content "here" (meaning to one of the sink devices connected directly to the set-top), or to a list of compatible sinks discovered elsewhere on the network. Figure 11 shows one view of the system.

An area we have explored in some depth is the potential of interoperability frameworks such as Obje to support easier collaboration among users, including not only information sharing, but also device sharing. For example, the Casca tool was created to support ongoing, small-group collaboration, allowing users to publish access to files and devices from their laptops into a shared space, thus making them available to others who are "members" of that space [Edwards et al. 2002b]. Another application that uses Obje to support collaboration is the Sharing Palette [Voida et al. 2006], which provides the ability to push content to collaborators using a lightweight icon bar, as well as to establish sharing groups and publicly-accessible files and devices. Both of these tools leverage Obje to allow collaborators to not only files share, but also access devices and services; for example, these tools can allow a user to provide access to his or her webcam, to a restricted filespace, or a home printer to a collaborator: essentially, whatever Obje devices or services are available can be shared using these tools. Figure 12 shows both of tools.

Still another tool is a specialized application for meeting-room display control [Newman et al. 2007], shown in Figure 13, which allows easy networked discovery and control of displays, including projectors and plasma screens. This application *only* discovers devices that can accept viewable media types; in our environment, this typically includes devices such as projectors and plasma displays. Users can select a discovered display and mirror their laptop screens onto it. This tool has seen daily use around PARC in a number of meeting rooms for

Fig. 12. The image on the left shows Casca. The open area to the right of the window is a canvas into which files, services, and devices can be dragged and dropped, sharing them with the current members of that shared space. The image on the right shows the Sharing Palette, a lightweight interface for small group-oriented sharing.

Fig. 13. The Obje Display Mirror before and after a connection is made. On the left, the user sees a list of available screens. After connection, a control UI is presented to all connected users of the selected screen, allowing control of the display to be shared.

over a year. This interface limits the complexity seen by the user by restricting the user's view to the networks of appropriate peers only.

We believe that these tools point to the power of allowing open-ended device access *through* traditional applications, which may serve to contextualize users' needs better, and may help constrain the set of available options. For example, while the functionality provided by the meeting-room tool could be accessed through a generic browser (find and select the laptop mirror service, find and select the projector device, connect them, and so on), the meeting-room tool contextualizes this functionality to provide easier access by making assumptions

about use (only show display devices, assume that the laptop mirror service is running on the current machine).

8. SYSTEM EXPERIENCES AND REFLECTION

In addition to gaining experience about how users cope with open-ended opportunities for device use, we have also gained a number of important insights regarding our underlying system design. The major motivation for the Obje architecture was to provide an infrastructure for unplanned, opportunistic interoperation between networked devices and to explore programmatic patterns for defining these devices. As such, we believe that our metrics for success rest both on usability issues, such as described above, and on design issues, that is, whether we can support the sorts of powerful, open-ended device use we had envisioned. In our research, issues such as performance, development complexity, and so on took a tertiary position.

In order to evaluate the "correctness" of our design—how well it can accommodate a range of devices and application needs—we built a wide variety of Obje-enabled devices and applications, which required our architecture to cope with diverse performance, security and usability requirements, as well as a variety of data types. To date, we have developed a dozen distinct applications using the architecture, support for roughly fifty devices and services (including a number of "proxy" services for accessing legacy devices, including Bluetooth and USB; media-oriented components such as DVD and CD players, music and video libraries, and networked displays; software services such as IM, gateways and RSS feeds; and others), and over a dozen typehandlers for various media (including image, audio, and video formats, as well as rich content types such as Powerpoint). As our core interfaces have stabilized, we have been able to build both extremely generic and extremely domain-specific applications with relative ease.

The applications described earlier have proven able to appropriate new devices and content types seamlessly as they appear on the network. For example, while the Casca collaborative tool was not specifically written to support video conferencing, it acquires that functionality as soon as members of a group share cameras, speakers, and microphones. In general, each of these applications demonstrates the value of an interoperability approach that combines fixed programmatic interfaces (for compatibility) with mobile code (for runtime extensibility). Each of our applications is able to interact with new types of devices that become available on the network, including devices that appeared after the applications were written. These applications acquire new behaviors from the devices that appear, extending their abilities to interact with new media types, communicate using new protocols, and present custom user interfaces. In view of the fact that we have not yet encountered applications that caused our model to break, our experience to date has encouraged us that the Obje architecture can support a wide range of device types.

Our meta-interface approach does, however, come with some performance costs due to the use of mobile code. When a connection between devices is initiated, there is startup overhead as bootstrap occurs and granules are sent over

the wire. However, once the necessary granule arrives at a peer, the performance is comparable to the hard-coded case. For example, we have successfully transmitted DVD-quality MPEG2 video data across a (wired) IP network with no visible performance issues once the video begins streaming. Startup latency has been acceptable for interactive applications (in the fraction of a second range), even for complex granules such as in the case of the MPEG2 decoder used in our set-top box application.

Much of the complexity associated with creating Obje-enabled devices is mitigated through toolkits we have created to allow developers to wrap existing devices and services with Obje functionality. Generally, relatively minimal code is required to add Obje functionality to a software service. For example, an existing text-to-speech service required roughly 50 lines of glue code to make it a native Obje service. Creating software proxies for a nonObje device is generally fairly easy, as long as the device can be accessed via software libraries or documented protocols. The hardest case, of course, is directly running Obje on embedded devices, which requires porting the bootstrap protocol implementation directly to the device. We have successfully ported the core Obje software stack to a range of devices, including a home gateway device and a number of mobile phones and PDAs.

While we have not performed any formal evaluations to measure the complexity of supporting new devices and services with our toolkit, we have been able to observe a number of interns and external collaborators use the code, and have been generally pleased with the ability of developers other than the system creators to use it. As an example, we have found that undergraduate interns have been able to build relatively sophisticated Obje-native services (such as an initial version of the extensible public display system described in Black et al. [2003]) in only a few days.

After a number of iterations of the system, our core platform is relatively tuned, small, and portable, and runs on a range of laptop, desktop, and server systems (including Windows, Linux, MacOS X, and Solaris), as well as mobile and embedded platforms. For example, a very portable, Java-based implementation of the Obje runtime platform is well under a megabyte, and runs on both "heavyweight" desktop virtual machines (such as Java SE version 6) as well as "lightweight" embedded virtual machines (including the CreMe virtual machine,[4] a small VM for embedded devices that equates roughly to JDK 1.1.8); we have also created a more tuned version of around 200KB designed to run on mobile phones under the J2ME CDC Profile.

Pervasive use of mobile code is a key architectural feature of Obje; however, mobile code brings with it a number of challenges and caveats. The issue of security is obviously a paramount concern, as discussed earlier. However, another challenge of dealing with mobile code is to ensure that the execution environment expected by a granule exists at a receiving device. While this challenge certainly exists for platform-specific native code (which, modulo emulation facilities, is bound to particular CPU and operating system types), we note that it is also true of supposedly platform-independent code as well. For example,

[4]http: / / www.nsicom.com.

Java code has dependencies on libraries being installed on the intended host, correct version matches, and so forth.

Our current solution is to make high-level declarations of the dependencies that must be satisfied for a given custom object to run. These dependencies are communicated to potential hosts via the Obje bootstrap protocol, where they can be evaluated by the host to see if the custom object may be executed by it. Declarations are simply in the form of strings ("Java 1.3.1 + javax.comm" or "x86 Windows XP") that must be understood by the intended recipient in order for the code to be executable.

This technique is robust enough for our current uses, but could obviously be significantly expanded. We are investigating a more complex format for the declaration of dependencies that must be satisfied in order for mobile code to run, as well as some automated tools for extracting such dependencies. This more structured format should allow more robust arbitration of code dependencies.

9. CONCLUSIONS AND FUTURE DIRECTIONS

This article has described Obje, a system designed to support ad hoc interoperation of devices and services on a network through the use of a fixed set of interfaces to dynamically extend behavior. The architectural approaches of the system have evolved in response to our experiences with applications and with users.

We make no claims that the set of mechanisms presented here is the only way one can address ad hoc interoperation. Obje embodies one approach, which we have found to work well for the set of applications and users we have investigated. Other approaches are certainly possible, and may better support certain applications. In general, however, we have been pleased by the range of applications we can support with the system, as well as the richness of interactions among devices.

We believe that the primary challenges posed by our approach come from the user-experience perspective. In particular, the overarching question is whether users can accomplish their goals with a system that potentially provides less application support than has been traditional. We believe that new UI techniques can help to maintain the usability that might be missing otherwise. The Obje system demonstrates the potential benefit of removing application constraints from decisions about interoperability; the challenge now is to address the user experience issues that this opening affords. We believe that the trade-off in power versus usability is worth it already; further work on service composition, and moving toward creating lightweight, task-oriented representations of device state will only shift the balance further in favor of architectural approaches that can allow this sort of ad hoc interoperability.

ACKNOWLEDGMENTS

We wish to thank the large number of people who have contributed to Obje either through its development or its use. In particular we wish to acknowledge

the contributions of Mark Howard, Andy Vyrros, Alyssa Glass, and Karen Marcelo to the development of the system; Diana Smetters, Dirk Balfanz, and Hao-Chi Wong for their work on the security aspects of Obje; our interns Shahram Izadi, Julie Black, Jason Hong, and Niels Castle Anderson; Nicolas Ducheneaut and Beki Grinter for invaluable guidance in our study designs; and PARC management for their support over the years.

REFERENCES

BALFANZ, D., SMETTERS, D. K., STEWART, P., AND WONG, H. C. 2002. Talking to strangers: Authentication in ad hoc wireless networks. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'02)*.

BANDELLONI, R., MORI, G., AND PATERNO, F. 2005. Dynamic generation of web migratory interfaces. In *Proceedings of the Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI)*.

BHARAT, K. A. AND CARDELLI, L. 1995. Migratory applications. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'95)*. ACM, New York, 133–142.

BLACK, J. A., EDWARDS, W. K., NEWMAN, M. W., SEDIVY, J. Z., AND SMITH, T. F. 2003. Supporting extensible public display systems with Speakeasy. In *Public and Situated Displays: Social and Interactional Aspects of Shared Display Technologies*, K. O'Hara, et al. Eds., Kluwer Academic, Amsterdam.

BLUETOOTH CONSORTIUM. 2001. Specification of the Bluetooth System, Version 1.1 Core. http://www.bluetooth.com.

BORENSTEIN, N. AND FREED, N. 1992. MIME (multipurpose internet mail extensions): Mechanisms for specifying and describing the format of internet messages. Internet RFC 1341.

BOX, D., EHNEBUSKE, D., KAKIVAYA, G., LAYMAN, A., MENDELSOHN, N., NIELSEN, H. F., THATTE, S., AND WINER, D. 2000. W3C note: Simple object access protocol (SOAP) 1.1. World Wide Web Consortium.

CHAKRABORTY, D. AND JOSHI, A. 2001. Dynamic service composition: state-of-the-art and research directions. Tech. rep.TR-CS-01-19, CSEE, University of Maryland, Baltimore.

DUCHENEAUT, N., SMITH, T., BEGOLE, J., NEWMAN, M. W., AND BECKMANN, C. 2006. The Orbital browser: Composing Ubicomp services using only rotation and selection. In *Proceedings of the ACM Conference on Human-Factors in Computing Systems (CHI'06)*. ACM, New York, 321–326.

EDWARDS, W. K., NEWMAN, M. W., SEDIVY, J. Z., SMITH, T. F., AND IZADI, S. 2002a. Challenge: Recombinant computing and the Speakeasy approach. In *Proceedings of the 8th ACM International Conference on Mobile Computing and Networking (MOBICOM)*. ACM, New York.

EDWARDS, W. K., NEWMAN, M. W., SEDIVY, J. Z., SMITH, T. F., BALFANZ, D., SMETTERS, D. K., WONG, H. C. AND IZADI, S. 2002b. Using Speakeasy for ad hoc peer-to-peer collaboration. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*. ACM, New York, 256–265.

EDWARDS, W. K., BELLOTTI, V., DEY, A. K., AND NEWMAN, M. W. 2003. Stuck in the middle: The challenges of user-centered design and evaluation of infrastructure. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. ACM, New York, 297–304.

EDWARDS, W. K., NEWMAN, M. W., SEDIVY, J. Z. AND SMITH, T. F. 2004. Supporting serendipitous integration in mobile computing environments. *Int. J. Hum. Comput. Stud. 60*, 666–700.

EDWARDS, W. K., NEWMAN, M. W., SEDIVY, J. Z., AND SMITH, T. F. 2005. An extensible set- top box platform for home media applications. *IEEE Trans.Consumer Electron. 51*, 4, 1175–1181.

EDWARDS, W. K. 2006. Discovery systems in ubiquitous computing. *IEEE Pervasive Comput. 5*, 2, 70–77.

FOX, A., GOLDBERG, I., GRIBBLE, S. D., LEE, D. C., POLITO, A., AND BREWER, E. A. 1998. Experience with Top Gun Wingman: A proxy-based graphical web browser for the 3com Palm Pilot. In *Proceedings of the Middleware Conference*.

GELERNTER, D. 1985. Generative communication in Linda. *ACM Trans. Program. Languages Syst. 7*, 1, 80–112.

GONG, L. 1999. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, MA.

GRIBBLE, S. D., WELSH, M., VON BEHREN, J. R., BREWER, E. A., CULLER, D., BORISOV, N., CZERWINSKI, S., GUMMADI, R., HILL, J., JOSEPH, A., KATZ, R. H., MAO, Z. M., ROSS, S. AND ZHAO, B. 2001. The Ninja architecture for robust internet-scale systems and services. *Comput. Netw. 35*, 4, 473–497.

HARMONIA INC. 2000. User interface modeling language 2.0 draft specification. http://www.uiml.org/specs/uiml2/index.htm.

HAN, R., PERRET, V., AND NAGHSHINEH, M. 2000. WebSplitter: A unified XML famework for multi-device collaborative web browsing. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*. ACM, New York, 221–230.

HODES, T. AND KATZ, R. H. 1999. A document-based framework for internet application control. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS)*. 59–70.

HUANG, A. C., LING, B. C., BARTON, J. AND FOX, A. 2001. Making computers disappear: appliance data services. In *Proceedings of the 7th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*.

HUMBLE, J., CRABTREE, A., HEMMINGS, T., AKESSON, K.-P., KOLEVA, B., RODDEN, T. AND HANSSON, P. 2003. Playing with the bits: User-configuration of ubiquitous domestic environments. In *Proceedings of the 5th International Conference on Ubiquitous Computing.*

INTERNET ENGINEERING TASK FORCE (IETF). 2005. Zeroconf Working Group. http://www.zeroconf.org.

JOHANSON, B., FOX, A., AND WINOGRAD, T. 2002. The interactive workspaces project: experiences with ubiquitous computing rooms. *IEEE Pervasive Comput. 1*, 2, 71–78.

JERONIMO, M. AND WEAST, J. 2003. *UPnP Design by Example*. Intel Press.

KICIMAN, E., MELLOUL, L., AND FOX, A. 2001. Towards zero-code service composition. In *Proceedings of the Workshop on Hot Topics on Operating Systems (HOTOS)*.

KICZALES, G., DES RIVIERES, J., AND BOBROW, D. G. 1991. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA.

KINDBERG, T. AND BARTON, J. 2001. A web-based nomadic computing system. *Comput. Netw. 35*, 4, 443–456.

KINDBERG, T. AND FOX, A. 2002. System software for ubiquitous computing. *IEEE Pervasive Comput. 1*, 1, 70–81.

LEE, K., LAMARCA, A., AND CHAMBERS, C. 2003. HydroJ: Object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, New York.

LISKOV, B. AND WING, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. 16*, 6, 1811–1841.

MACKAY, W. E. 1990. Patterns of sharing customizable software. In *Proceedings of the Conference on Computer Supported Cooperative Work*. ACM, New York.

MACKAY, W. E. 1991. Triggers and barriers to customizing software. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'91)*. ACM, New York, 153–160.

MAO, Z. M. AND KATZ, R. H. 2002. Achieving service portability using self-adaptive data paths. *IEEE Comm. 40*, 1, 108–114.

NEWMAN, M. W., IZADI, S., EDWARDS, W. K., SEDIVY, J. Z., AND SMITH, T. F. 2002a. User interfaces when and where they are needed: An infrastructure for recombinant computing. In *Proceedings of the 15th ACM Symposium on User Interface Software and Technology (UIST'02)*. ACM, New York.

NEWMAN, M. W., SEDIVY, J. Z., EDWARDS, W. K., SMITH, T. F., MARCELO, K., NEUWIRTH, C. M., HONG, J. I. AND IZADI, S. 2002b. Designing for serendipity: Supporting end-user configuration of ubiquitous computing environments. In *Proceedings of the Designing Interactive Systems Conference (DIS'02)*.

NEWMAN, M., DUCHENEAUT, N., EDWARDS, W. K., SEDIVY, J., AND SMITH, T. 2007. Supporting the unremarkable: Experiences with the Obje display mirror *Personal Ubiquitous Comput*. (DOI:10.1007/s00779-006-0117-0). To appear.

NEWMAN, M., ELLIOTT, A., AND SMITH, T. F. 2008. Providing an integrated user experience of networked media, devices, and services through end-user composition. In *Proceedings of the International Conference on Pervasive Computing (PERVASIVE'08)*.

NICHOLS, J., MYERS, B. A., HIGGINS, M. HUGHES, J., HARRIS, T. K., ROSENFELD, R., AND PIGNOL, M. 2002. Generating remote control interfaces for complex appliances. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'02)*. ACM, New York, 161–170.

OBJECT MANAGEMENT GROUP. 1995. CORBA: The common object request broker architecture, Rev. 2.0.

OCKERBLOOM, J. 1998. Mediating among diverse data formats. Tech.rep., CMU-CS-98-10, Carnegie Mellon University.

OLSEN, D. R., JEFFRIES, S., NIELSEN, T., MOYES, W., AND FREDERICKSON, P. 2000. Crossmodel interaction with XWeb. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*. ACM, New York, 191–200.

OMOJOKUN, O. AND DEWAN, P. 2003. A high-level and flexible framework for dynamically composing networked devices. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)* .

PONNEKANTI, S. R., LEE, B., FOX, A., HANRAHAN, P., AND WINOGRAD, T. 2001. ICrafter: A service framework for ubiquitous computing environments. In *Proceedings of the UBICOMP Conference*. 56–75.

PONNEKANTI, S. R., AND FOX, A. 2004. Interoperability among independently evolving web services. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*. 331–351.

REKIMOTO, J., AYATSUKA, Y., KOHNO, M., AND OBA, H. 2003. Proximal Interactions: A direct manipulation technique for wireless networking. In *Proceedings of the INTERACT Conference*. Richardson, T., Stafford-Fraser, Q., Wood, K. and Hopper, A. 1998. Virtual network computing. *IEEE Internet Comput. 2,* 1.

ROSE, M. 2001. RFC 3080: The blocks extensible exchange protocol core. Internet Engineering Task Force (IETF).

SMETTERS, D. K. AND GRINTER, R. E. 2002. Moving from the design of usable security technologies to the design of useful secure applications. In *Proceedings of the ACM New Security Paradigms Workshop*. ACM, New York.

SMETTERS, D. K., BALFANZ, D., DURFEE, G., SMITH, T., AND LEE, K. 2006. Instant matchmaking: Simple secure virtual extensions to ubiquitous computing environments. In *Proceedings of the 8th International Conference on Ubiquitous Computing (UBICOMP)*. Lecture Notes in Computer Science, vol. 4206, Springer, Berlin, 477–494.

STAJANO, F. AND ANDERSON, R. J. 1999. The resurrecting duckling: Security issues for ad hoc wireless networks. In *Proceedings of the 7th Security Protocols Workshop*. Lecture Notes in Computer Science, vol. 1796, Springer, Berlin, 172–194.

SUN MICROSYSTEMS. 1999. Jini discovery and join specification.

SWINDELLS, C., INKPEN, K. M., DILL, J. C., AND TORY, M. 2002. Use that there! Pointing to establish device identity. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*. ACM, New York.

TRUONG, K. N., HUANG, E. M., AND ABOWD, G. D. 2004. CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. In *Proceedings of the 6th International Conference on Ubiquitous Computing (UBICOMP)*. 143–160.

UPNP FORUM. 2005. MediaServer V 1.0 and Media Renderer V 1.0. http://www.upnp.org/standardizeddcps/mediaserver.asp.

UNIVERSAL SERIAL BUS IMPLEMENTER'S FORUM. 2000. Universal serial bus revision 2.0 specification.

VENNERS, B. 2005. The ServiceUI API specification, version 1.1a. http://www.artima.com/jini/serviceui/Spec.html.

VOIDA, S., EDWARDS, W. K., NEWMAN, M. W., GRINTER, R. E., AND DUCHENEAUT, N. 2006. Share and share alike: Exploring the user interface affordances of file sharing. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. ACM, New York.

WALDO, J. 1999. The Jini architecture for network-centric computing. *Comm. ACM*, 76–82.

WEISER, M. AND BROWN, J. S. 1996. Designing calm technology. http://powergrid.electriciti.com/1.01.

WHITTEN, A. AND TYGAR, J. D. 1999. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the 9th USENIX Security Symposium*. 23–26.

WING, J. M. AND OCKERBLOOM, J. 2000. Respectful type converters. *IEEE Trans. Softw. Engin. 28*, 7, 579–593.

WOLLRATH, A., RIGGS, R., AND WALDO, J. 1996. A distributed object model for the Java system. *USENIX Comput. Syst. 9*.

Received June 2005; revised March 2007; accepted July 2007 by Prasun Dewan