IEEE pervasive COMPUTING
MOBILE AND UBIQUITOUS SYSTEMS
www.computer.org/pervasive

Bringing Network Effects to Pervasive Spaces

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

Vol. 4, No. 3
July–September 2005

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

IEEE COMPUTER SOCIETY

IEEE COMMUNICATIONS SOCIETY

© 2005 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

For more information, please see www.ieee.org/portal/pages/about/documentation/copyright/polilink.html.

Standards & Emerging Technologies

Editor: Sumi Helal ■ University of Florida ■ helal@cise.ufl.edu

Bringing Network Effects to Pervasive Spaces

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

In the January–March 2005 installment of this department, Sumi Helal made a case for developing new approaches to interoperability. In doing so, he hit a strong chord with our research group at the Palo Alto Research Center (PARC).

Back in 2000, we began exploring the human experience of richly networked and interconnected environments. We envisioned several application scenarios, which we then tried to realize by creating a range of networked services and digital devices. We wanted these services and devices to freely interconnect with each other and recombine to provide new functionality. However, we quickly realized the impediments—many of which Helal identified in his article—inherent in actually trying to create such environments. Furthermore, we recognized that the most dire impediment isn't battery life, processing power, or low-level network connectivity. It's that devices won't work together unless you explicitly program each one to talk to every other type of device it might encounter.

So, in addition to exploring the human experience of living in richly networked settings, we ended up exploring the interoperability that might enable such settings to exist in the first place. In the course of our research, we developed the *Obje Interoperability Framework*, a middleware technology that lets networked applications and services coordinate with each another—even when they know almost nothing about one other.

THE COMBINATORIAL COMPLEXITY OF ONGOING INTEGRATION

Currently, developers must program devices to recognize the specific protocols, standards, data formats, and operations of all the peer device types they expect their devices to encounter. The balance of work in this model is fundamentally wrong. If one new device type appears on the network, we must update all existing devices. Integrating device types becomes increasingly difficult as the number of existing devices in need of upgrades, software installations, or replacement continues to grow.

Devices won't work together unless you explicitly program each one to talk to every other type of device it might encounter.

This property—the combinatorial complexity of ongoing integration—exists in virtually every widely available infrastructure for interoperability today. In the Universal Plug and Play world, for instance, a device might be programmed to communicate with peers that implement the UPnP MediaRenderer profile. These same devices, however, would have to be reprogrammed to communicate with UPnP printers or with new types of UPnP devices that might appear in the future (including new versions of the MediaRenderer profile).

This need for detailed, device type-specific programming, built into each peer device, is at the root of the problems Helal identified:

- *Nonscalable integration.* Every existing device must be updated to work with new device types.
- *Closed-world assumptions.* Getting two device types to work together requires explicit programming, making adding support for new devices difficult.
- *Fixed-point concepts.* Because developers program against the set of standards known at development time, they can't accommodate entirely new classes of devices that don't resemble existing standards.

The end result is today's limited interoperability, which in turn limits the network effects key to many visions of ubiquitous computing. We need to move toward a model of computing where we can simply plug new device types into the network and all existing peers on the network will be able to use

1536-1268/05/$20.00 © 2005 IEEE ■ *Published by the IEEE CS and IEEE ComSoc*

STANDARDS & EMERGING TECHNOLOGIES

them. In other words, we need to move away from a model of combinatorial complexity toward one of constant complexity—where integrating each new device is just as easy as integrating the one before it.

RECOMBINANT COMPUTING

The Obje infrastructure (formerly known as Speakeasy) uses *recombinant computing* to let devices interact without previous approaches' limitations and strictures. Recombinant computing recognizes that, fundamentally, communication between any two parties is predicated on shared knowledge about the terms of communication. As in human communication, we must have some shared language to be able to understand each other.

In the case of systems such as UPnP, this shared knowledge takes the form of the underlying platform assumptions (TCP/IP, HTTP, XML, or SSDP), as well as the profiles that describe each device type's capabilities. The problem with this "profile" approach is that it requires prior knowledge of details specific to each individual device type. A printer profile, for example, represents information specific to printers and thus is useless for other device types—information about stapling, duplicates, and ink color. Because these profiles are so specific, we usually can't reuse them for similar devices. (For example, in UPnP, "MediaServers" and "Scanners" are different device profiles, even though both produce media.) This specificity also requires changing a profile if we can't retrofit a new device type's capabilities to work with the existing profile. (For example, a new type of printer appears with watermarking capabilities; we must express access to these capabilities in the profile for client devices to be able to access them.)

The profile approach grows increasingly fragile over time. Profiles must expand to encompass new device capabilities and proliferate to cover new device types. With Obje, we moved away from the profile-oriented model and instead used a few simple abstractions that remain constant and that we assume all peers on the network understand.

Obje, for example, represents only four major classes of device capabilities. A device can

- connect to another device,
- provide metadata about itself,
- be controlled, and
- provide references to other devices.

In this approach, we code a device's software against the generic abstractions the infrastructure defines rather than the specifics of any peer device. So, devices can interact with new device types that appear on the network as long as they represent their functionality using this base set of abstractions. In essence, the Obje infrastructure maps the various devices' underlying functionality on the network into a simplified form designed to facilitate easier interconnection and ensure compatibility with future devices.

Obje lets devices declare the formats they can provide, and recipients can declare which formats they can execute.

Normally, such a generic and semantically neutral set of capabilities would be hugely restrictive. Our approach overcomes such restrictions by introducing the notion of *dynamic extensibility*. Rather than describing how a device communicates with its peer, these interfaces describe how a device *acquires new behavior* from its peer that allows communication. We use the agreed-upon interfaces coded into software only as a generic bootstrapping mechanism to acquire code that implements the specific communication mechanisms any given device needs.

For example, an Obje-enabled display device (such as a projector) can support the ability to be connected to another Obje-enabled device (such as a PDA, networked video camera, or laptop). Obje neither dictates nor requires that the connection be established over HTTP, Real-Time Transfer Protocol, or any other fixed protocol. Nor does it dictate that only JPEGs, GIFs, or MPEG-2 must be transferred. Rather, it facilitates a mechanism by which the two devices can exchange new behaviors that let them communicate with each other, using whatever protocols are appropriate for the data being exchanged. They also let the recipient render the received data. In particular, the specific details of how to receive or render data need not be built into the display device up front; rather, the receiving device acquires these details from the sending device at runtime.

Our approach establishes the minimal set of development-time agreements required to defer all other agreements until runtime. It then delivers these agreements in the form of mobile code, which can extend a recipient's behavior to make it compatible with a new peer. Devices that enter the network carry the code they need to let peers communicate with them. This extension of behavior is generally transient—the device adopts the new behavior only when communicating with the peer that provided the behavior. More importantly, because the new devices' developers are responsible for creating the mobile code bundle, the burden of bringing a new device onto the network rests with them instead of with the existing peers.

MINIMIZING AGREEMENTS

Our work on the Obje infrastructure has revolved around defining a set of development-time agreements that are as minimal as possible (thus imposing the fewest requirements on Obje-enabled devices), while allowing the greatest possible flexibility. Thus, mobile code can't extend the behavior of peers in completely arbitrary ways. Rather, the platform dictates how a peer can extend a device (in the form of oper-

ations to acquire new code) and dictates the interfaces that any received code must integrate (the mechanisms for object instantiation, operation invocation, and so forth). These mechanisms offer runtime extensibility to new data transfer protocols, new content-type handling mechanisms, new control interfaces, and new discovery protocols.

The other fixed agreements that must be in place concern the underlying platform assumptions necessary to acquire the code in the first place. The Obje bootstrapping protocol is a thin remote invocation protocol layered atop the Blocks Extensible Exchange Protocol. BEEP is a TCP/IP-based framing and messaging protocol that's easily portable and provides extensible security provisioning. Thus, as part of the fixed agreement that Obje devices must implement to participate, they must be TCP/IP-enabled and contain an implementation of our lightweight messaging protocol, defined on BEEP.

Likewise, when a device transmits mobile code, it must be in a format that the recipient can execute. Obje lets devices declare the formats they can provide (Java bytecodes and native x86 dynamic library code, for instance), and recipients can declare which formats they can execute. These optional agreements might limit the interoperability achievable in practice, but they make the core platform agnostic with regard to the executable format.

USER-SUPPLIED SEMANTICS

An important implication in this move away from a profile-oriented interoperability infrastructure is that devices in abstraction-based infrastructures might not understand the semantics of the peers they encounter. For example, a program coded to use UPnP MediaRenderers presumably knows what a MediaRenderer does: its developer understands the semantics of displaying content, pausing, fast forwarding, and so forth, and when it might be appropriate to do so. This semantic knowledge is encoded into the software itself. In an Obje world, on the other hand, the programming on each device only tells the device how to interact with peers using the abstract mechanisms outlined earlier. So, an Obje-enabled PDA might be able to communicate with a new device, but it might not know what that device does, or when or whether it's appropriate to communicate with it.

Such semantic ignorance is necessary in a world designed for open-ended interoperability. If each device must know the semantics of the peers it encounters, then we're back in the same closed-world situation as before. An open-world approach asks devices to interact with peers about which they know little. Responsibility for determining appropriate interactions between devices shifts from the developer to the end user. The infrastructure's role is merely to ensure that interoperability is possible; the user must decide which devices make sense together.

The infrastructure's role is merely to ensure that interoperability is possible; the user must decide which devices make sense together.

Such a world will be characterized by generic tools that let end-users compose and configure devices within a space. Users will be able to enter a pervasive space, quickly assess device capabilities, and then assemble the devices into a desired configuration.

We've implemented the current version of the Obje Framework (version 4) in Java. It supports both Java and native mobile code and can be implemented in a variety of languages and platforms. It also runs on several platforms, including Windows, Mac OS X, Linux, and Windows PocketPC.

Leveraging the Obje Framework, we have been able to return to our focus of developing pervasive spaces to explore the human experience of pervasive computing. We currently have a half-dozen Obje-enabled applications and several dozen interoperable components using the framework, including home media components (Obje-enabled DVD players, speakers, and media libraries) and office components (projectors, plasma displays, and whiteboard capture services).

We've deployed many of these components throughout PARC to help us understand how users work, live, and play in pervasive spaces—how they use the devices around them and recombine them into new configurations. We're now developing tools to support easy end-user composition by letting end users create assemblages of devices and services that can support them in their tasks. P

W. Keith Edwards is an associate professor at the Georgia Tech College of Computing. Contact him at keith@cc.gatech.edu.

Mark W. Newman is a research scientist at the Palo Alto Research Center and a doctoral student in computer science at the University of California Berkeley. Contact him at mnewman@parc.com.

Jana Z. Sedivy was a research scientist at PARC until 2005, and she contributed to the development of the Obje Interoperability Framework. Contact her at janasedivy@yahoo.com.

Trevor F. Smith is a member of the ubicomp group in the computing science laboratory at the Palo Alto Research Center. Contact him at tfsmith@parc.com.