System Software for Ubiquitous Computing

The authors identify two key characteristics of ubiquitous computing systems, physical integration and spontaneous interoperation. They examine how these properties affect the design of ubicomp software and discuss future directions.

Tim Kindberg
Hewlett-Packard Laboratories

Armando Fox
Stanford University

Ubiquitous computing, or ubicomp, systems designers embed devices in various physical objects and places. Frequently mobile, these devices—such as those we carry with us and embed in cars—are typically wirelessly networked. Some 30 years of research have gone into creating distributed computing systems, and we've invested nearly 20 years of experience in mobile computing. With this background, and with today's developments in miniaturization and wireless operation, our community seems poised to realize the ubicomp vision.

However, we aren't there yet. Ubicomp software must deliver functionality in our everyday world. It must do so on failure-prone hardware with limited resources. Additionally, ubicomp software must operate in conditions of radical change. Varying physical circumstances cause components routinely to make and break associations with peers of a new degree of functional heterogeneity. Mobile and distributed computing research has already addressed parts of these requirements, but a qualitative difference remains between the requirements and the achievements. In this article, we examine today's ubiquitous systems, focusing on software infrastructure, and discuss the road that lies ahead.

Characteristics of ubiquitous systems

We base our analysis on physical integration and spontaneous interoperation, two main characteristics of ubicomp systems, because much of the ubicomp vision, as expounded by Mark Weiser and others,[1] either deals directly with or is predicated on them.

Physical integration

A ubicomp system involves some integration between computing nodes and the physical world. For example, a *smart coffee cup*, such as a Media-Cup,[2] serves as a coffee cup in the usual way but also contains sensing, processing, and networking elements that let it communicate its state (full or empty, held or put down). So, the cup can give colleagues a hint about the state of the cup's owner. Or consider a smart meeting room that senses the presence of users in meetings, records their actions,[3] and provides services as they sit at a table or talk at a whiteboard.[4] The room contains *digital furniture* such as chairs with sensors, whiteboards that record what's written on them, and projectors that you can activate from anywhere in the room using a PDA (personal digital assistant).

Human administrative, territorial, and cultural considerations mean that ubicomp takes place in more or less discrete *environments* based, for example, on homes, rooms, or airport lounges. In other words, the world consists of ubiquitous systems rather than "the ubiquitous system." So, from physical integration, we draw our Boundary Principle:

Ubicomp system designers should divide the ubicomp world into environments with *boundaries* that demarcate their content. A clear *system boundary criterion*—often,

Figure 1. The ubiquitous computing world comprises environments with boundaries and components appearing in or moving between them.

but not necessarily, related to a boundary in the physical world—should exist. A boundary should specify an environment's scope but doesn't necessarily constrain interoperation.
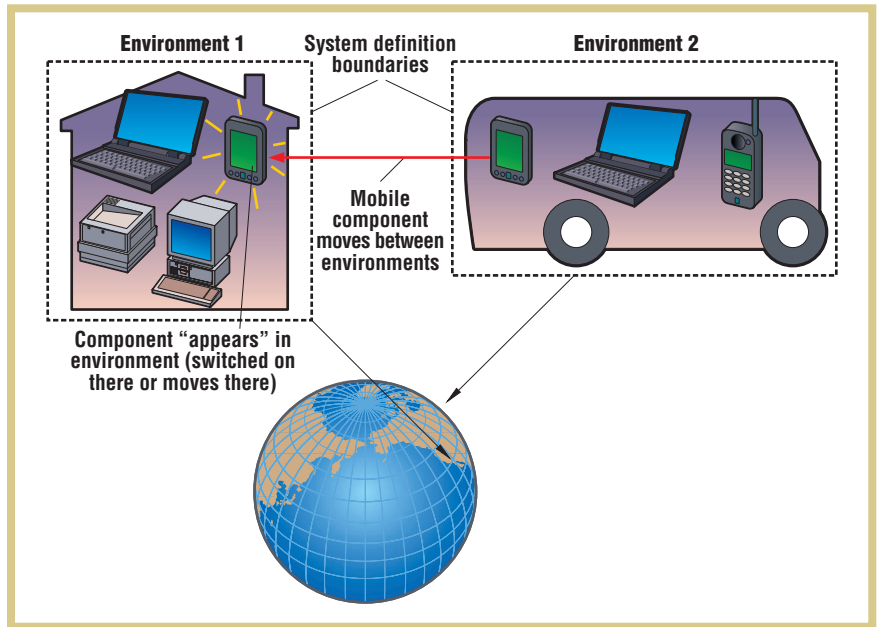
Spontaneous interoperation

In an environment, or *ambient*,[5] there are *components*—units of software that implement abstractions such as services, clients, resources, or applications (see Figure 1). An environment can contain infrastructure components, which are more or less fixed, and *spontaneous* components based on devices that arrive and leave routinely. Although this isn't a hard and fast distinction, we've found it to be useful.

In a ubiquitous system, components must spontaneously interoperate in changing environments. A component interoperates spontaneously if it interacts with a set of communicating components that can change both identity and functionality over time as its circumstances change. A spontaneously interacting component changes partners during its normal operation, as it moves or as other components enter its environment; it changes partners without needing new software or parameters.

Rather than a de facto characteristic, spontaneous interoperation is a desirable ubicomp feature. Mobile computing research has successfully addressed aspects of interoperability through work on adaptation to heterogeneous content and devices, but it has not discovered how to achieve the interoperability that the breadth of functional heterogeneity found in physically integrated systems requires.

For a more concrete definition, suppose the owners of a smart meeting room propose a "magic mirror," which shows those facing it their actions in the meeting. Ideally, the mirror would interact with the room's other components from the moment you switch it on. It would make spontaneous associations with all relevant local sources of information about users. As another example, suppose a visitor from another organization brings his PDA into the room and, without manually configuring it in any way, uses it to send his presentation to the room's projector.

We choose *spontaneous* rather than the related term *ad hoc* because the latter tends to be associated with networking—ad hoc networks[6] are autonomous systems of mobile routers. But you cannot achieve spontaneous interoperation as we have defined it solely at the network level. Also, some use ad hoc to mean infrastructure-free.[7] However, ubicomp involves both infrastructure-free and infrastructure-enhanced computing. This includes, for example, spontaneous interaction between small networked sensors such as particles of *smart dust*,[8] separate from any other support, or our previous example of a PDA interacting with infrastructure components in its environment.

From spontaneous interoperation, we draw our Volatility Principle:

> You should design ubicomp systems on the assumption that the set of participating users, hardware, and software is highly dynamic and unpredictable. Clear invariants that govern the entire system's execution should exist.

The mobile computing community will recognize the Volatility Principle in theory, if not by name. However, research has concentrated on how individual mobile components adapt to fluctuating conditions. Here, we emphasize that ubicomp adaptation should involve more than "every component for itself" and not restrict itself to the short term; designers should specify system-wide invariants and implement them despite volatility.

Examples

The following examples help clarify how physical integration and spontaneous interoperation characterize ubicomp systems. Our previous ubicomp example of the magic mirror demonstrates physical integration because it can act as a conventional mirror and is sensitive to a user's identity. It spontaneously interoperates with components in any room.

Nonexamples of ubicomp include

- *Accessing email over a phone line from a laptop*. This case involves neither physical integration nor spontaneous interoperation; the laptop maintains the same association to a fixed email server. This exemplifies a physically mobile system: it can operate in various physical environments but only because those environments are equally transparent to it. A truly mobile (although not necessarily ubiquitous) system engages in spontaneous interactions.[9]

- *A collection of wirelessly connected laptops at a conference*. Laptops with an IEEE 802.11 capability can connect spontaneously to the same local IEEE 802.11 network, assuming no encryption exists. Laptops can run various applications that enable interaction, such as file sharing. You could argue that discovery of the local network is physical integration, but you'd miss the essence of ubicomp, which is integration with that part of the world that has a
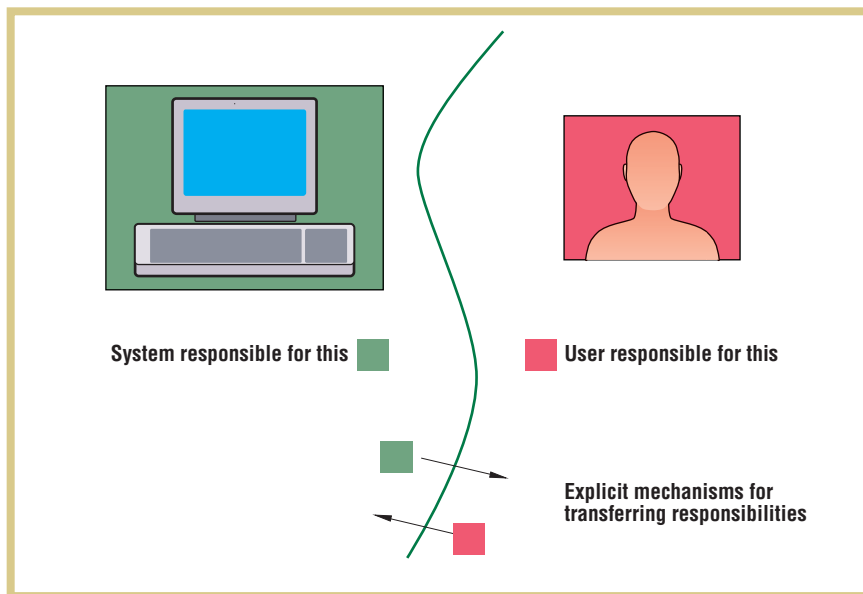
nonelectronic function for us. As for spontaneous interaction, realistically, even simple file sharing would require considerable manual intervention.

Borderline ubicomp examples include

- *A smart coffee cup and saucer.* Our smart coffee cup (inspired by but different from the MediaCup) clearly demonstrates physical integration, and it demonstrates a device that you could only find in a ubiquitous system. However, if you constructed it to interact only with its corresponding smart saucer according to a specialized protocol, it would not satisfy spontaneous interoperation. The owner couldn't use the coffee cup in another environment if she forgot the saucer, so we would have localization instead of ubiquitous functionality.
- *Peer-to-peer games.* Users play games such as Pirates![10] with portable devices connected by a local area wireless network. Some cases involve physical integration because the client devices have sensors, such as proximity sensors in Pirates! Additionally, some games can discover other players over the local network, and this dynamic association between the players' devices resembles spontaneous interaction. However, such games require preconfigured compo-

nents. A more convincing case would involve players with generic game-playing "pieces," which let them spontaneously join local games even if they had never encountered them before.

- *The Web.* The Web is increasingly integrated with the physical world. Many devices have small, embedded Web servers,[11] and you can turn objects into *physical hyperlinks*—the user is presented with a Web page when it senses an identifier on the object.[12] Numerous Web sites with new functionality (for example, new types of e-commerce sites) spring up on the Web without, in many cases, the need to reconfigure browsers. However, this lacks spontaneous interoperation—the Web requires human supervision to keep it going. The "human in the loop" changes the browser's association to Web sites. Additionally, the user must sometimes install plug-ins to use a new type of downloaded content.

Software challenges

Physical integration and spontaneous interoperation have major implications for software infrastructure. These ubicomp challenges include a new level of component interoperability and extensibility, and new dependability guarantees, including adaptation to changing environments, tolerance of routine failures or failurelike con-

ditions, and security despite a shrunken basis of trust. Put crudely, physical integration for system designers can imply "the resources available to you are sometimes highly constrained," and spontaneous interoperation means "the resources available to you are highly dynamic but you must work anywhere, with minimal or no intervention." Additionally, a ubicomp system's behavior while it deals with these issues must match users' expectations of how the physical world behaves. This is extremely challenging because users don't think of the physical world as a collection of computing environments, but as a collection of places[13] with rich cultural and administrative semantics.

The semantic Rubicon

Systems software practitioners have long ignored the divide between system- and human-level semantics. Ubicomp removes this luxury. Little evidence exists to suggest that software alone can meet ubicomp challenges, given the techniques currently available. Therefore, in the short term, designers should make clear choices about what system software will not do, but humans will. System designers should pay careful, explicit attention to what we call the *semantic Rubicon* of ubiquitous systems (see Figure 2). (The historical Rubicon river marked the boundary of what was Italy in the time of Julius Caesar. Caesar brought his army across it into Italy, but only after great hesitation.) The semantic Rubicon is the division between system and user for high-level decision-making or physical-world semantics processing. When responsibility shifts between system and user, the semantic Rubicon is crossed. This division should be exposed in system design, and the criteria and mechanisms for crossing it should be clearly indicated. Although the semantic Rubicon might seem like a human-computer interaction (HCI) notion,

especially to systems practitioners, it is, or should be, a ubicomp systems notion.

Progress report

Many ubicomp researchers have identified similar challenges but pursued them with quite different approaches; we offer a representative sampling and comparison of these approaches. Our discussion focuses on these areas, which appear common to Weiser's and others' ubicomp scenarios:

- *Discovery*. When a device enters an environment, how does mutual discovery take place between it and other available services and devices, and among which ones is interaction appropriate?
- *Adaptation*. When near other heterogeneous devices, how can we use the device to display or manipulate data or user interfaces from other devices in the environment?
- *Integration*. What connects the device's software and the physical environment, and what affects the connection?
- *Programming framework*. What does it mean to write the traditional programming exercise "Hello World" for a ubicomp environment: Are discovery, adaptation, and integration addressed at the application level, middleware–OS level, language level, or a combination of these?
- *Robustness*. How can we shield the device and user from transient faults and similar failures (for example, going out of network range) that will likely affect a ubicomp environment?
- *Security*. What can we let the device or user do? Whom does it trust and how does authentication take place? How can we minimize threats to privacy?

Discovery and interaction

Ubiquitous systems tend to develop accidentally over the medium-to-long term,[14] that is, in piecemeal as users integrate new devices into their physical environments and as they adopt new usage models for existing devices. You can't reboot the world, let alone rewrite it, to introduce new functionality. Users shouldn't be led into a disposable physical world just because they can't keep up with software upgrades. All this suggests that ubicomp systems should be incrementally extensible.

Spontaneous interoperation makes shorter-term demands on our components' interoperability. As we defined a spontaneously interoperating system, devices can interact with partners of varying functionality over time. We do not mean arbitrarily varying functionality: For example, it's not clear how a camera could meaningfully associate with a coffee cup. Our challenge is to bring about spontaneous interaction between devices that conform to widespread models of interoperation—as wide as is practicable.

Consider a component that enters an environment (see Figure 1). To satisfy spontaneous interoperation, the component faces these issues:

- *Bootstrapping*. The component requires a priori knowledge of addresses (for example, multicast or broadcast addresses) and any other parameters needed for network integration and service discovery. This is largely a solved problem. Protocols exist for joining underlying networks, such as IEEE 802.11. You can dynamically assign IP addresses either statefully, using DHCP (dynamic host configuration protocol),[15] or statelessly in IPv6.[16]
- *Service discovery*. A service discovery system dynamically locates a service instance that matches a component's requirements. It thus solves the *association problem*: Hundreds or even thousands of devices and components might exist per cubic meter; with which of these, if any, is it appropriate for the arriving component to interact?
- *Interaction*. Components must conform to a common interoperation model to interact.

Additionally, if the arriving component implements a service, the components in the service's environment might need to discover and interact with it. Service discovery and interaction are generally separable (although the Intentional Naming System[17] combines them).

Service discovery. Address allocation and name resolution are two examples of local services that an arriving component might need to access. Client components generally require an a priori specification of required services, and corresponding service components must provide similar specifications to be discoverable. Several systems perform service discovery.[17–21] Each provides syntax and vocabulary for specifying services. The specification typically consists of attribute–value pairs such as *serviceType=printer* and *type=laser*.

A challenge in ubicomp service description is avoiding overspecification. Consider a wireless-enabled camera brought into a home. If the camera contains a printing service description, a user could take a picture and send it from the camera to a matching printer without using a PC. But one issue is brittleness: Both camera and printer must obey the same vocabulary and syntax. Association can't take place if the camera requests *serviceType=printing* when the printer has *service=print*. Agreement on such details is a practical obstacle, and it's debatable whether service specification can keep up with service development.

Furthermore, lost opportunities for association arise if devices must contain specifications of all services with which they can interact. Suppose our example home contains a digital picture frame. A camera that can send an image for printing should be able to send the image to a digital frame. But, if the camera doesn't have a specification for *serviceType=digitalFrame*, the opportunity passes.

Other designs, such as Cooltown,[21] tried to use more abstract and much less detailed service specifications to get around such problems. For example, a camera could send its images to any device that consumes images of the same encoding (such as JPEG) without needing to know the specific service.

But abstraction can lead to ambiguity. If a camera were to choose a device in the house, it would potentially face many data consumers with no way of discriminating between them. Also, it's not clear how a camera should set the target device's parameters, such as image resolution, without

a model of the device. Users, however, tend to be good at associating devices appropriately. So, we observe:

> System designers must decide the human's role to resolve tension between interoperability and ambiguity.

Additionally, the Boundary Principle tells us that the definition of *here* is important for meaningful discovery results. For example, a client might not be interested in services in Japan if he has just arrived in a smart house in London. Figure 1 shows various environments with a dotted line marking the boundary around service components that we deem to be in the same environment as the arriving device.

A new challenge for ubiquitous systems arises because adaptation must often take place without human intervention, to achieve what Weiser calls *calm* computing.

In one approach, clients might sense their current location's name or position and use that in their service specifications to a global discovery service. But most service discovery systems are local, of the *network discovery* type. They primarily define *here* as the collection of services that a group, or multicast, communication can reach. The discovery service listens for query and registration messages on a multicast address that all participating components possess a priori. These messages' scope of delivery typically includes one or more local connected subnets, which they can efficiently reach with the underlying network's multicast or broadcast facilities.

The problem here is that the approach puts discovery firmly on the systems side of the semantic Rubicon, but the very simplicity of subnet multicast makes it blind to human issues such as territory or use conventions. It is unlikely, for example, that the devices connected to a particular subnet are in a meaningful place such as a room. Using physically constrained media such as infrared, ultrasound, or 60-GHz radio (which walls substantially attenuate) helps solve territorial issues, but it does not

solve other cultural issues. Another alternative is to rely on human supervision to determine the scope of each instance of the discovery service.[22]

Interaction. After associating to a service, a component employs a programming interface to use it. If services in a ubicomp environment arbitrarily invented their own interfaces, no arriving component could be expected to access them. Jini lets service-access code and data, in the form of a Java object, migrate to a device. However, how can the software on the device use the downloaded object without a priori knowledge of its methods?

Event systems[18, 23] or tuple spaces[18,24–26] offer alternative approaches. These interactions share two common features: the system interface comprises a few fixed operations, and the interactions are data-oriented.

In event systems, components called *publishers* publish self-describing data items, called *events* or *event messages*, to the local event service. That service matches the data in a published event to specifications that components called *subscribers* have registered. When an event is published, the event service forwards it to all subscribers with a matching subscription. The only interaction methods an event system uses are *publish*, *subscribe*, and *handle* (that is, handle an event).

A tuple space provides a repository of data tuples. Again, any component interacting via a tuple space needs to know only three operations. (A fourth operation, *eval*, exists in some tuple space systems for process or thread creation.) Components can *add* tuples to a tuple space, or they can *take* tuples out or *read* them without removing them.

In each case, components interact by sharing a common service (a tuple space or an event service); they don't need direct

knowledge of one another. However, these data-oriented systems have shifted the burden of making interaction consistent onto data items—namely, events and tuples. If one component publishes events or adds tuples of which no other component has a priori knowledge, interaction can fail. So, we observe:

> Data-oriented interaction is a promising model that has shown its value for spontaneous interaction inside the boundaries of individual environments. It seems that this requires *ubiquitous data standardization* for it to work across environment boundaries.

Adaptation

Several reasons exist for why ubicomp implies dealing with limited and dynamically varying computational resources. Embedded devices tend to be small, and limits to miniaturization mean constrained resources. Additionally, for devices that run on batteries, a trade-off exists between battery life and computational prowess or network communication. For extreme examples of ubicomp such as smart dust, these constraints become an order of magnitude more severe than for components such as PDAs, which previous research has considered resource-impoverished.

Furthermore, the available resources tend to vary dynamically. For example, a device that has access to a high-bandwidth wireless network such as IEEE 802.11b in one environment might find itself with only a low-bandwidth wide-area connection in another—a familiar problem in mobile computing. A new challenge for ubiquitous systems arises because adaptation must often take place without human intervention, to achieve what Weiser calls *calm* computing.[27] In this new environment, we examine possible extensions of existing mobile computing techniques: transformation and adaptation for content and the human interface.

Content. Embedding computation in or linking general-purpose computing devices with the physical world implies heterogeneity across the devices, including devices embedded in and those brought to the environment by users (for example, lap-
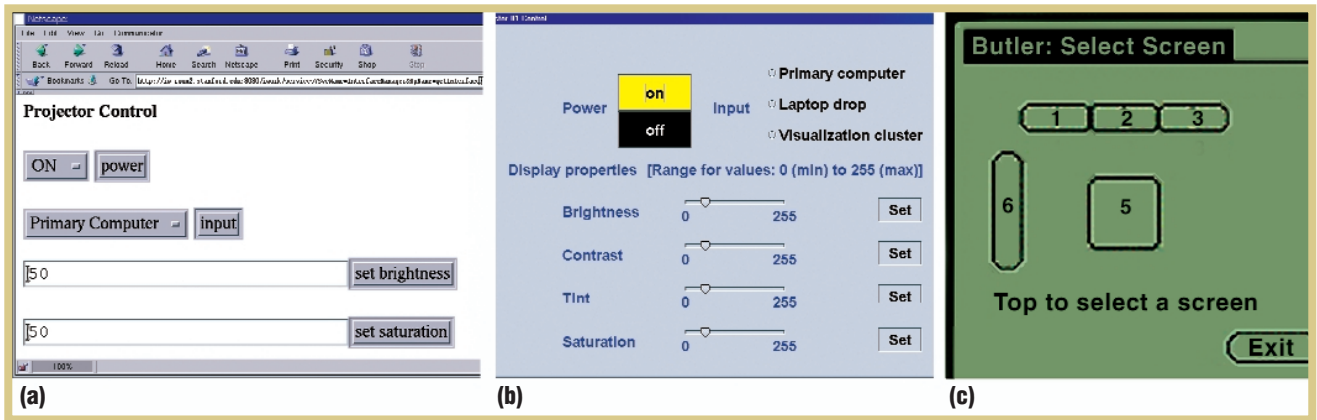
Figure 3. Spontaneously generated user interfaces to control lights and displays in a meeting room. The same high-level markup was used to generate device-specific UIs for (a) a desktop HTML browser, (b) a Java-equipped device, and (c) a Palm device.

tops). Mobile computing research successfully addressed content adaptation for resource-poor devices at both the OS and application levels. Pioneering work such as the Coda file system explored network disconnection and content adaptation within the context of a specific application, namely the file system, handling all such adaptation in the operating system to enable application transparency. Coda's successor, Odyssey,[29] and other, later work on adapting Web content for slow networks and small devices[30,31] moved some responsibility to the application layer, observing that because the OS is rarely familiar with the semantics of the application's data, it is not always in the best position to make adaptation decisions; this reflected the thinking of application-level framing.[32]

Adaptation in ubicomp is quantitatively harder. Instead of adapting a small fixed set of content types to a variety of device types (1-to-*n*), we must potentially adapt content among *n* heterogeneous device types (*n*-to-*n*). Fortunately, the general approaches mobile computing research has explored have immediate applicability. For example, the Stanford Interactive Workspaces' *smart clipboard* can copy and paste data between incompatible applications on different platforms.[33] It builds on well-known mobile computing content adaptation, the main difference being that the smart clipboard must transparently invoke the machinery whenever the user performs a copy-and-paste operation. A more sophisticated but less general approach, *semantic snarfing*, as implemented in Carnegie Mellon's Pebbles project, captures content from a large display onto a small display and attempts to

emulate the content's underlying behaviors.[34] For example, snarfing a pop-up menu from a large display onto a handheld lets the user manipulate the menu on her handheld, relaying the menu choice back to the large display. This approach tries harder to do "the right thing," but rather than relying on a generic transformation framework, it requires special-case code for each content type (such as menus and Web page content) as well as special-case client-server code for each pair of desired devices (Windows to Palm, X/Motif to Windows CE). Nonetheless, both approaches demonstrate the applicability of content transformation to the ubicomp environment. In general, we observe:

> The content adaptation approaches and technology developed in mobile computing immediately pertain to ubicomp, but they solve only a part of the overall problem. We must still address such issues as discovery and robustness, and we must find a way to apply the adaptation using mechanisms invisible to the user.

The human interface. Earlier, we introduced the commonly quoted ubicomp scenario of a user entering an environment and immediately accessing or controlling various aspects of the environment through her PDA. To achieve such a scenario requires separating user interfaces (UIs) from their applications—a well-understood problem in desktop systems. In ubicomp, however, the goal might be to move the human interface to a physically different device chosen on the fly. Based on previous on-the-fly transformations applied to this problem, we can distinguish four levels of client intelligence. At the lowest

level, a client such as VNC[35] displays the bits and collects user input, without knowledge of UI widget semantics. At the next level, the client manipulates a higher-level, usually declarative description of the interface elements and has the intelligence to render them itself. X Windows and its low-bandwidth derivatives—for example, LBX (Low-Bandwidth X)—do this, although in these systems, the device displaying the UI runs what is confusingly called a server. Smarter clients also do their own geometry management and some local UI interaction processing, so not every interaction requires a roundtrip to the server (see Figure 3); Tcl/Tk and JavaScript-enhanced Web pages are examples.

A noteworthy variant involves delivering an HTML UI (based on forms and HTML widgets) via HTTP to any device that can host a Web browser, thus removing the need to reconfigure the device for different interfaces. This important special case arises as a result of the Web's phenomenal success. However, HTML provides only limited expressiveness for a UI. Also, HTML and HTTP address delivering, rendering, and letting the user interact with the UI, but they do not address how to determine what belongs in it or how UI elements associate with the applications or devices they control.

The next level is represented by Todd Hodes and his colleagues,[36] who proposed a Tcl/Tk framework in which controllable entities export Tcl-code descriptions, from which a client can generate its UI. The descriptions are declarative and high level ("This control is a Boolean toggle," "This control is a slider with range 0 to 255 in

increments of 1"), letting the geometry and the generated interface's visual details vary among platforms. The Stanford Interactive Workspaces project has successfully generalized Hodes's approach: The Interface Crafter framework[4] supports a level of indirection of specialized per-device or per-service *interface generators* and a generic interface transformation facility. In this system, the high-level service descriptions permit adaptation for nontraditional UI modalities such as voice control.

Finally, a fully generalized mobile code facility, such as Jini, lets the client download and execute arbitrary code or pseudocode that implements the interface and communication with the service. Because the protocol between a Jini service and its UI is *opaque*—embedded in the code or pseudocode—you usually can't enlist a different user agent to interpret and render the UI. You also can't realize the UI on a device not running a Java Virtual Machine or on one with nontraditional I/O characteristics (for example, a voice-controlled or very small glass device). We conclude:

> Applying transformation to the UI can be a powerful approach to achieving spontaneous interaction in ubicomp, provided we express the UI in terms that enable extensive manipulation and transformation.

Integration with the physical world

To integrate computing environments (which are virtual by definition) with the physical world, we need low-level application programming interfaces that let software deal with physical sensors and actuators, and a high-level software framework that lets applications sense and interact with their environment, including physical sensors and actuators. Traditional device drivers offer APIs that are too low-level, because application designers usually think of functionality at the level of widgets in standard toolkits such as Tcl/Tk, Visual Basic, and X/Motif. A potentially more useful low-level API is a Phidget,[37] a GUI widget element whose state and behaviors correspond to those of a physical sensor or actuator. For example, querying a servomotor phidget returns the motor's angular position, and setting the Phidget's angular value causes the motor to step to the given position. You can directly incorporate Phidgets into Visual Basic programs using the popular Visual Basic WYSIWYG drag-and-drop UI editor. At a higher level, the Context Toolkit framework[38] provides applications with a context widget software abstraction, which lets an application access different types of context information while hiding how the information was sensed or collected. For example, one type of context widget provides information on the presence of a person in a particular location, without exposing how it collected

It's important to leverage existing applications because of large investments in the applications, their data, and knowledge of how to use them.

the information. The Context Toolkit's middleware contribution is its ability to expose a uniform abstraction of (for example) location tracking, hiding the details of the sensing system or systems used to collect the information.

Location sensing and tracking figure prominently in several projects. The early Active Badge work[39] and the more recent Sentient Computing effort[40] use infrared, radio, and ultrasound to track physical objects that users carry or wear. Knowing the users' locations and identities enables location tracking and novel behaviors for existing devices, such as a camera that knows who it's photographing. On the other hand, EasyLiving[41] and the MIT Intelligent Room[42] use location tracking as one of several elements that help infer or disambiguate a user action. For example, if a user requests "Show me the agenda," the system can respond in a context-sensitive manner if it can determine the user's identity and location. Such approaches lead to potentially impressive applications, although they should make it clear on which side of the semantic Rubicon decisions are being made at the point of integration with the user's physical world. Such systems should make it easy to identify where the decisions are being made at the point of integration with the user's physical world.

Several groups have investigated ways of linking physical entities directly to services using identification technologies. For example, Roy Want and his colleagues at Xerox Parc[43] augmented books and documents by attaching radio frequency identification tags to them and presenting the electronic version to users who scanned them with handheld devices. Jun Rekimoto and Yuji Ayatsuka[44] attached symbols specially designed for digital camera capture to physical objects, to link them to electronic services. Hewlett-Packard Labs' Cooltown project focuses on associating Web resources (*Web presences*) with physical entities. For example, a visitor to a museum can obtain related Web pages automatically over a wireless connection by reading identifiers associated with the exhibits. The identifier might come from a barcode or a short-range infrared beacon. Although the Cooltown infrastructure has primarily targeted direct human interaction, work is in progress on interoperation between Web presences using XML over HTTP to export query and update operations.[45]

Programming frameworks

What does it mean to write "Hello World" for a ubicomp environment? A framework for programming such environments might address the ubicomp challenges in applications, the operating system or middleware, and the development language. Some environments might use combinations of these.

Support for legacy applications and commodity OSs has always been a systems research challenge. Although we have stressed the need for incremental extensibility in any new system, legacy support also implies accommodating existing applications and OSs. It's important to leverage existing applications because of large investments in the applications, their data, and knowledge of how to use them. It's

important to leverage OSs without modification because OSs are fragile, most users are reluctant or insufficiently sophisticated to apply OS patches, and OS functionality is generally regarded as beyond application writers' control. (This latter point can be especially true in ubicomp, as seen in specialized embedded OSs such as the TinyOS used in smart dust.[46]) Although some researchers choose to design entirely new systems at the expense of legacy support, we do not always have this option in ubicomp. The Volatility Principle tells us that ubicomp environments change incrementally and continuously over time: today's new systems are tomorrow's legacy systems, and given innovation's current pace, legacy systems are accreting faster than ever. Therefore, our approaches should include support for legacy systems. The Stanford Interactive Workspaces project has taken this approach, providing only an application coordination mechanism via a tuple space[26] and delegating state storage, sensor fusion for context determination, and so forth to other components at the application level. This results in more loosely coupled pieces, but makes the basic infrastructure easy to port to new and legacy devices, encouraging rapid device integration as an aspect of incremental evolution.

Conversely, the MIT Intelligent Room project's Metaglue framework provides Java-specific extensions for freezing object state in a persistent database and expressing functional connections between components. (For example, you can express that one component relies on the functionality of another, and the system provides resource brokering and management to satisfy the dependency at runtime.) In Metaglue, the component coordination system is also the language and system for building the applications. The researchers chose this approach, which forgoes legacy support, as a better fit for the project's focus—namely, applying AI techniques to determine the intent of a user's action (for example, by sensing identity via image analysis from a video feed and position relative to landmarks in the room). Although AI techniques can make valuable contributions, such work must clearly define on which side of the semantic Rubicon a particular functionality resides. An important question will be whether the Metaglue framework is sufficiently expressive and provides abstractions at the appropriate level to facilitate this.

Other places to provide programming support for addressing the ubicomp challenges are middleware and the OS. When mobile computing faced similar challenges in breaking away from the "one desktop machine per fixed user" model of computing, researchers introduced middleware as one approach. We define middleware as services provided by a layer in between the operating system and the applications. Middleware usually requires only minimal changes to existing applications and OSs (hence its name). The ActiveSpaces project at the University of Illinois, Urbana-Champaign, is developing a large framework, Gaia,[47] based on distributed objects and a software object bus that can connect objects from different frameworks (Corba, DCOM, Enterprise JavaBeans). Gaia, a middleware OS, views an ActiveSpace and its devices as analogous to a traditional OS with the resources and peripherals it manages. The framework thus provides more tightly integrated facilities for sensor fusion, quality-of-service-aware resource management, code distribution, adaptive and distributed rendering, and security. The ActiveSpaces project chooses to deemphasize extensive support for existing systems and devices.

Robustness and routine failures

Ubiquitous systems, especially those using wireless networking, see a radical increase of "failure" frequency compared to a wired distributed system. Some of these failures are not literal failures but unpredictable events from which it is similarly complicated to recover. For example, physical integration often implies using batteries with a relatively short mean time to failure due to battery exhaustion. Wireless networks, with limited range and prone to interference from nearby structures, afford much less reliable communication than wireline networks. In spontaneous interoperation, associations are sometimes gained and lost unpredictably as, for example, when a device suddenly leaves an environment.

Although the distributed systems literature contains techniques for fault-tolerant computing, they are often based on resource redundancy rather than scarcity. Furthermore, the common assumption in distributed systems that failures are relatively rare could lead to expensive or ungraceful recovery, contrary to the expectation of calm behavior for ubiquitous systems.

Two widely accepted truths about dependability are that you can't add it to a system after the fact but must design it in,[48] and that it is ultimately an end-to-end property.[49] Design choices made at lower layers of a system can either facilitate or hinder the sound engineering of dependability at higher system layers or in the project's later stages.

In ubicomp systems, various transient failures can actually characterize steady-state operation. The Internet protocol community and the distributed systems community have considered such scenarios. We look to both for techniques specifically designed to function under the assumption that "failure is a common case." The common tie is the underlying assumption that recovering from frequent transient failures basically entails being always prepared to reacquire lost resources.

Expiration-based schemes and soft state.
Consider the PointRight system,[50] in which a single mouse and keyboard can control a collection of otherwise-independent displays in a ubicomp environment. When a user enters or leaves a ubicomp environment that has PointRight, the user's device could register with a centralized directory that tracks the room's screen geometry. But this directory is a single point of failure. If it fails and has not saved its data to stable storage, it must notify all devices to reregister when the directory comes back up. Furthermore, if a device dies or fails to deregister before leaving the environment, it creates inconsistency between the directory contents and the environment's true state.

An alternative solution requires each available device or service to send a peri-

odic advertisement announcing its presence or availability to the directory service, which collects these advertisements and expires them when a new advertisement arrives or after a designated timeout period. Should the directory service fail, new advertisements will repopulate it when it restarts; hence, the directory itself constitutes soft state. Should a device fail, it will stop sending advertisements, and once the device's previous advertisement has expired, the directory will no longer list the device.

A detailed analysis of how to choose a timeout to balance the resulting "window of potential inconsistency" with the network and computation capacity consumed by advertisements is available elsewhere.[51] Of course, it might be more practical to store nonchanging information such as physical room geometry in a fixed database for reliability and high performance. Microsoft takes this approach for describing the geometry of EasyLiving-aware rooms. We propose the following suggestion for practitioners:

> When failure is a common case, identify what critical static or dynamic state you must persistently store and consider reconstructable soft state for the rest.

Separating failure-free and failure-prone operations. In a dynamically changing environment, some operations can fail while others must necessarily succeed. The one.world middleware separates application code into operations, which can fail, and logic, which does not fail except under catastrophic circumstances.[23] Generally, an operation is anything that might require allocating or accessing a potentially nonlocal resource whose availability is dynamic, such as file or network I/O. For robustness, a process must always be prepared to rediscover and reacquire lost resources. For example, opening a file on a remote server creates a binding between the local filehandle and the remote file. If the application thread is migrated to another host, the file might still be available on the server, but you'll need to establish a new binding on the new host. The application must provide code that deals with such conditions when doing opera-

tions that can fail. one.world automatically provides for some common cases, such as retrying idempotent operations a finite number of times. So, we observe:

> Because not all operations are equally likely to fail, clearly indicate which ones are more likely to fail because of dynamism and required spontaneous interoperation, so developers can provide more effective error handling.

Group communication for "free" indirection. You can use group communication to provide a level of indirection that helps rediscover lost resources. Although approaches that rely on group communication have been criticized for their poor scaling, in ubicomp, we might be able to exploit the Boundary Principle. For example, a meeting room only accommodates a finite number of persons, because effective meetings must necessarily be limited in size; a corporate campus has a security and administrative boundary that separates it from the rest of the world; and so on. In fact, systems that aim to provide location-dependent services might be thwarted by the *reverse scalability* problem: wireless group communication's typical scope often exceeds the size of the locale in which the service will operate. So, we observe:

> Although ubicomp projects should certainly address scalability, the Boundary Principle can confound the attempt. Sometimes observing the Boundary Principle can help, by enabling solutions that favor robustness over scalability. Other times, observing the principle can introduce complications, as is the case when a physically range-limited networking technology still fails to capture other (cultural or administrative) important boundaries.

Security

We described how components enter into spontaneous associations in a ubiquitous system and interact with the components they discover. However, how do we protect components from one another? Certain services in a smart room, for example, need protection from devices belonging to visitors. Similarly, data and devices brought into an environment, such as users' PDAs or particles of smart dust, require protection from hostile devices nearby.

Looked at from a higher level, users

require security for their resources and, in many cases, privacy for themselves.[52] Mobile computing has led to an understanding of vulnerabilities such as the openness of wireless networks.[53] But physical integration and spontaneous interoperation raise new challenges, requiring new models of trust and authentication as well as new technologies.

Trust. In ubiquitous systems, trust is an issue because of spontaneous interoperation. What basis of trust can exist between components that can associate spontaneously? Components might belong to disparate individuals or organizations and have no relevant a priori knowledge of one another or a trusted third party. Fortunately, physical integration can work to our advantage—at least with an appropriate placement of the semantic Rubicon. Humans can make judgments about their environments' trustworthiness,[7] and the physical world offers mechanisms for bootstrapping security based on that trust. For example, users might exchange cryptographic keys with their environment or each other over a physically constrained channel such as short-range infrared, once they have established trust.

Security for resource-poor devices. Physical integration impacts security protocols. Particles of smart dust and other extremely resource-poor devices do not have sufficient computing resources for asymmetric (public key) encryption—even when using elliptic curve cryptography[54]—and protocols must minimize communication overheads to preserve battery life. For example, Spins (security protocols for sensor networks) provides security guarantees for the data that smart dust particles exchange in a potentially hostile environment.[55] They use only symmetric-key cryptography, which, unlike asymmetric-key cryptography, works on very low-power devices. But this does not address what has been called the "sleep deprivation torture attack" on battery-powered nodes:[56] an attacker can always deny service by jamming the wireless network with a signal that rapidly causes the devices to run down their batteries.

Access control. Another problem in ubicomp systems is basing access control on authenticating users' identities. Physical integration makes this awkward for users because knowing an identified user's whereabouts raises privacy issues. Spontaneous interoperation makes it problematic for environments, which have to integrate a stream of users and devices that can spontaneously appear and disappear. It can be advantageous to issue time-limited capabilities to devices that have spontaneously appeared in an environment, rather than setting up access control lists. Capabilities could be issued on the basis of dynamically established trust such as we described earlier and used without explicitly disclosing an identity. Also, they reduce the need for system-wide configuration and avoid communication with a central server.[19] Because capabilities resemble physical keys, a security component can enable, for example, a visitor's PDA to use a soft-drinks machine without communicating with that machine.

Location. Physical integration has several implications for security through location. Customizing services to a location can result in a loss of privacy for identified individuals. However, location-aware computing doesn't have to require user tracking. Users, rather than the system, can establish their own locations.[57] Moreover, even if the system learns the user's location, it does not follow necessarily that it can correlate that location with any personal data. In a ubicomp system, locations can be tied to temporary identifiers and keys, and some cases require no identity criteria. In *location authentication*, the system establishes a component's physical location, for example, as a criterion for providing services. So, an Internet cafe might provide a walk-up-and-print service to anyone who is on the cafe's premises. It doesn't matter who they are, the cafe cares only about where they are. Physically constrained channels, such as ultrasound, infrared, and short-range radio transmission, particularly at highly attenuated frequencies, have enabled work on protocols that prove whether clients are where they claim to be.[58,59]

New patterns of resource-sharing. We commonly use intranet architectures to protect resources, using a firewall that cleanly separates resources inside and outside an organization. Mobile ambients,[5] a model of firewalled environments components that can be crossed in certain circumstances, more closely address mobile computing requirements than those of ubicomp. Physical integration works against the firewall model because it entails sharing "home" devices with visitors. Spontaneous interaction makes extranet technologies unsuitable, because developers intend them for long-term relationships with outside users. Work on securing discovery services[19] and access to individual services[60] tends to assume the opposite of the firewall model: that locals and visitors go through the same security interface to access resources, although the former might require more convenient access.

Evidently, a more convenient and fine-grained model of protected sharing is required. As an example, Weiser envisioned devices you can use temporarily as personal devices and then return for others to use. The "Resurrecting Duckling" paper suggests how a user might do this.[56] The user *imprints* the device by transmitting a symmetric key to it over a physically secure channel, such as by direct contact. Once imprinted, the device only obeys commands from other components that prove possession of the same key until it receives instructions to return to the imprintable state.

C learly, much more research is needed. Consider the common scenario of a user who walks into an environment for the first time and "finds" and uses a printer.

Even this seemingly easy scenario raises subtle issues. How do you know if it's culturally OK to use a particular printer? (It might be for the boss's use, even though it's in a shared space on a shared network.) Which printers are conveniently located? How should we define ontologies for such things as "color printer"?

We are not aware of a de facto solution to the printer example. At least, no solutions exist that we can use out of the lab. Too often, we only investigate interoperability mechanisms within environments instead of looking at truly spontaneous interoperation between environments. We suggest exploring *content-oriented programming*—data-oriented programming using content that is standard across boundaries—as a promising way forward. The Web demonstrates the effectiveness of a system that enables content standards to evolve, while users invoke processing

In some cases, the research community isn't realistic enough about either physical-world semantics' complexity or known mechanisms' inadequacy to make decisions when responding to exigencies.

through a small, fixed interface.

In some cases, the research community isn't realistic enough about either physical-world semantics' complexity or known mechanisms' inadequacy to make decisions when responding to exigencies, as in adaptation and failure recovery. Some problems routinely put forward are actually AI-hard. For example, similar to the Turing test for emulating human discourse, we could imagine a meeting test for context-aware systems: can a system accurately determine, as compared with our human sensibilities, when a meeting is in session in a given room?

To make progress, ubicomp research should concentrate on the case where a human is supervising well-defined ubicomp aspects—for example, by assisting in appropriate association or by making a judgment about a system's boundary. Once the ubicomp community can get that right—and we believe that human-supervised operation

represents as much as a 75 percent solution for ubicomp—it will be time to push certain responsibilities across the semantic Rubicon, back toward the system.

By the way, has anyone seen a printer around here? ▣

REFERENCES

1. M. Weiser, "The Computer for the 21st Century," *Scientific American*, vol. 265, no. 3, Sept. 1991, pp. 94–104 (reprinted in this issue, see pp. 19–25).

2. M. Beigl, H.-W. Gellersen, and A. Schmidt, "MediaCups: Experience with Design and Use of Computer-Augmented Everyday Objects," *Computer Networks*, vol. 35, no. 4, Mar. 2001, pp. 401–409.

3. G.D. Abowd, "Classroom 2000: An Experiment with the Instrumentation of a Living Educational Environment," *IBM Systems J.*, vol. 38, no. 4, Oct. 1999, pp. 508–530.

4. S.R. Ponnekanti et al., "ICrafter: A Service Framework for Ubiquitous Computing Environments," *Ubicomp 2001: Ubiquitous Computing*, Lecture Notes in Computer Science, vol. 2201, Springer-Verlag, Berlin, 2001, pp. 56–75.

5. L. Cardelli and A.D. Gordon, "Mobile Ambients," *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, vol. 1378, Springer-Verlag, Berlin, 1998, pp. 140–155.

6. C.E. Perkins, ed., *Ad Hoc Networking*, Addison-Wesley, Reading, Mass., 2001.

7. L. Feeney, B. Ahlgren, and A. Westerlund, "Spontaneous Networking: An Application-Oriented Approach to Ad Hoc Networking," *IEEE Comm. Magazine*, vol. 39, no. 6, June 2001, pp. 176–181.

8. J.M. Kahn, R.H. Katz, and K.S.J. Pister, "Mobile Networking for Smart Dust," *Proc. Int'l Conf. Mobile Computing and Networking* (MobiCom 99), ACM Press, New York, 1999, pp. 271–278.

9. R. Milner, *Communicating and Mobile Systems: The π calculus*, Cambridge Univ. Press, Cambridge, UK, 1999.

10. S. Björk et al., "Pirates! Using the Physical World as a Game Board," *Proc. Conf. Human-Computer Interaction* (Interact 01), IOS Press, Amsterdam, 2001, pp. 9–13.

11. G. Borriello and R. Want, "Embedded Computation Meets the World Wide Web," *Comm. ACM*, vol. 43, no. 5, May 1999, pp. 59–66.

12. T. Kindberg et al., "People, Places, Things: Web Presence for the Real World," *Proc. 3rd IEEE Workshop Mobile Computing Systems and Applications* (WMCSA 2000), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 19–28.

13. S. Harrison and P. Dourish, "Re-Place-ing Space: The Roles of Place and Space in Collaborative Systems," *Proc. ACM Conf. Computer-Supported Cooperative Work* (CSCW 96), ACM Press, New York, 1996, pp. 67–76.

14. W.K. Edwards and R.E. Grinter, "At Home with Ubiquitous Computing: Seven Challenges," *Ubicomp 2001: Ubiquitous Computing*, Lecture Notes in Computer Science, vol. 2201, Springer-Verlag, Berlin, 2001, pp. 256–272.

15. R. Droms, *Dynamic Host Configuration Protocol*, RFC 2131, Internet Engineering Task Force, www.ietf.org.

16. S. Thomson and T. Narten, *IPv6 Stateless Address Autoconfiguration*, RFC 1971, Internet Engineering Task Force, www.ietf.org.

17. W. Adjie-Winoto et al., "The Design and Implementation of an Intentional Naming System," *Proc. 17th ACM Symp. Operating System Principles* (SOSP 99), ACM Press, New York, 1999, pp. 186–201.

18. K. Arnold et al., "The Jini Specification," Addison-Wesley, Reading, Mass., 1999.

19. S.E. Czerwinski et al., "An Architecture for a Secure Service Discovery Service," *Proc. 5th Ann. ACM/IEEE Int'l Conf. Mobile Computing and Networks* (MobiCom 99), ACM Press, New York, 1999, pp. 24–35.

20. E. Guttman, "Service Location Protocol: Automatic Discovery of IP Network Services," *IEEE Internet Computing*, vol. 3, no. 4, July/Aug. 1999, pp. 71–80.

21. T. Kindberg and J. Barton, "A Web-Based Nomadic Computing System," *Computer Networks*, vol. 35, no. 4, Mar. 2001, pp. 443–456.

22. J. Barton, T. Kindberg, and S. Sadalgi, "Physical Registration: Configuring Electronic Directories Using Handheld Devices," to be published in *IEEE Wireless Comm.*, vol. 9, no. 1, Feb. 2002; tech. report HPL–2001–119, Hewlett-Packard, Palo Alto, Calif., 2001.

23. R. Grimm et al., "Systems Directions for Pervasive Computing," *Proc. 8th Workshop Hot Topics in Operating Systems* (HotOS VIII), 2001, pp. 128–132.

24. N. Carriero and D. Gelernter, "Linda in Context," *Comm. ACM*, vol. 32, no. 4, Apr. 1989, pp. 444–458.

25. N. Davies et al., "Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications," *Proc. Joint Int'l Conf. Open Distributed Processing and Distributed Platforms* (ICODP/ICDP 97), Chapman and Hall, Boca Raton, Fla., 1997.

26. B. Johanson and A. Fox, "Tuplespaces as Coordination Infrastructure for Interactive Workspaces," *Proc. Workshop Application Models and Programming Tools for Ubiquitous Computing* (UbiTools 01), 2001, http://choices.cs.uiuc.edu/UbiTools01/pub/08-fox.pdf.

27. M. Weiser and J. Brown, "Designing Calm Technology," *PowerGrid J.*, vol. 1, 1996.

28. J.J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Trans. Computer Systems*, vol. 10, no. 1, Feb. 1992, pp. 3–25.

29. B.D. Noble, M. Price, and M. Satyanarayanan, "A Programming Interface for Application-Aware Adaptation in Mobile Computing," *Proc. USENIX Symp. Mobile and Location-Independent Computing*, USENIX, Berkeley, Calif., 1995.

30. B. Zenel and D. Duchamp, "A General Purpose Proxy Filtering Mechanism Applied to the Mobile Environment," *Proc. 3rd Ann. ACM/IEEE Intl. Conf. Mobile Computing and Networking*, ACM Press, New York, 1997, pp. 248–259.

31. A. Fox et al., "Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives," *IEEE Personal Comm.*, Aug. 1998, pp. 10–19.

32. D.D. Clark and D.L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," *Computer Comm. Rev.*, vol. 20, no. 4, Sept. 1990.

33. E. Kiciman and A. Fox, "Using Dynamic Mediation to Integrate COTS Entities in a Ubiquitous Computing Environment," *Proc. 2nd Int'l Symp. Handheld and Ubiquitous Computing* (HUC2K), Lecture Notes in Computer Science, vol. 1927, Springer-Verlag, Berlin, 2000, pp. 211–226.

34. B.A. Myers et al., "Interacting at a Distance Using Semantic Snarfing," *Ubicomp 2001:*

Ubiquitous Computing, Lecture Notes in Computer Science, vol. 2201, Springer-Verlag, Berlin, 2001, pp. 305–314.

35. T. Richardson et al., "Virtual Network Computing," *IEEE Internet Computing*, vol. 2, no. 1, Jan./Feb. 1998, pp. 33–38.

36. T.D. Hodes et al., "Composable Ad Hoc Mobile Services for Universal Interaction," *Proc. Third Int'l Symp. Mobile Computing and Networking* (MobiCom 97), ACM Press, New York, 1997, pp. 1–12.

37. C. Fitchett and S. Greenberg, "The Phidget Architecture: Rapid Development of Physical User Interfaces," *Proc. Workshop Application Models and Programming Tools for Ubiquitous Computing* (UbiTools 01), 2001, http://choices.cs.uiuc.edu/UbiTools01/pub/17-fitchett.pdf.

38. D. Salber, A.K. Dey, and G.D. Abowd, "The Context Toolkit: Aiding the Development of Context-Enabled Applications," *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems* (CHI 99), ACM Press, New York, 1999, pp. 434–441.

39. R. Want et al., "The Active Badge Location System," *ACM Trans. Information Systems*, vol. 10, no. 1, Jan. 1992, pp. 91–102.

40. M. Addlesee et al., "Implementing a Sentient Computing System," *Computer*, vol. 34, no. 8, Aug. 2001, pp. 50–56.

41. B. Brumitt et al., "EasyLiving: Technologies for Intelligent Environments," *Proc. 2nd Int'l Symp. Handheld and Ubiquitous Computing* (HUC2K), Lecture Notes in Computer Science, vol. 1927, Springer-Verlag, Berlin, 2000, pp. 12–29.

42. M. Coen et al., "Meeting the Computational Needs of Intelligent Environments: The Metaglue System," *Proc. 1st Int'l Workshop Managing Interactions in Smart Environments* (MANSE 99), Springer-Verlag, Berlin, 1999.

43. R. Want et al., "Bridging Physical and Virtual Worlds with Electronic Tags," *Proc. 1999 Conf. Human Factors in Computing Systems* (CHI 99), ACM Press, New York, pp. 370–377.

44. J. Rekimoto and Y. Ayatsuka, "CyberCode: Designing Augmented Reality Environments with Visual Tags," *Proc. Designing Augmented Reality Environments*, ACM Press, New York, 2000, pp. 1–10.

45. D. Caswell and P. Debaty, "Creating Web Representations for Places," *Proc. 2nd Int'l Symp. Handheld and Ubiquitous Computing* (HUC2K), Lecture Notes in Computer Science, vol. 1927, Springer-Verlag, Berlin, 2000, pp. 114–126.

46. D.E. Culler et al., "A Network-Centric Approach to Embedded Software for Tiny Devices," *Proc. DARPA Workshop Embedded Software*, 2001.

47. M. Roman and R.H. Campbell, "GAIA: Enabling Active Spaces," *Proc. 9th ACM SIGOPS European Workshop*, ACM Press, New York, 2000.

48. B. Lampson, "Hints for Computer System Design," *ACM Operating Systems Rev.*, vol. 15, no. 5, Oct. 1983, pp. 33–48.

49. J.H. Saltzer, D.P. Reed, and D.D. Clark, "End-to-End Arguments in System Design," *ACM Trans. Computer Systems*, vol. 2, no. 4, Nov. 1984, pp. 277–288.

50. B. Johanson, G. Hutchins, and T. Winograd, *PointRight: A System for Pointer/Keyboard Redirection among Multiple Displays and Machines*, tech. report CS-2000-03, Computer Science Dept., Stanford Univ., Stanford, Calif., 2000.

51. S. Raman and S. McCanne, "A Model, Analysis, and Protocol Framework for Soft State-Based Communication," *Proc. Special Interest Group on Data Communication* (SIGCOMM 99), ACM Press, New York, 1999, pp. 15–25.

52. M. Langheinrich, "Privacy by Design: Principles of Privacy-Aware Ubiquitous Systems," *Ubicomp 2001: Ubiquitous Computing*, Lecture Notes in Computer Science, vol. 2201, Springer-Verlag, Berlin, 2001, pp. 273–291.

53. N. Borisov, I. Goldberg, and D. Wagner, "Intercepting Mobile Communications: The Insecurity of 802.11," *Proc. 7th Ann. Int'l Conf. Mobile Computing and Networks* (MobiCom 2001), ACM Press, New York, 2001, pp. 180–188.

54. N. Smart, I.F. Blake, and G. Seroussi, *Elliptic Curves in Cryptography*, Cambridge Univ. Press, Cambridge, UK, 1999.

55. A. Perrig et al., "SPINS: Security Protocols for Sensor Networks," *Proc. 7th Ann. Int'l Conf. Mobile Computing and Networks* (MobiCom 2001), ACM Press, New York, 2001, pp. 189–199.

56. F. Stajano and R. Anderson, "The Resurrecting Duckling: Security Issues for Ad Hoc Wireless Networks," *Security Protocols*, Lecture Notes in Computer Science, vol. 1796, Springer-Verlag, Berlin, 1999, pp. pp. 172–194.

57. N.B. Priyantha, A. Chakraborty, and H. Balakrishnan, "The Cricket Location-Support System," *Proc. 6th Ann. Int'l Conf. Mobile Computing and Networks* (MobiCom 2000), ACM Press, New York, 2000, pp. 32–43.

58. E. Gabber and A. Wool, "How to Prove Where You Are," *Proc 5th ACM Conf. Computer and Comm. Security*, ACM Press, New York, 1998, pp. 142–149.

59. T. Kindberg and K. Zhang, "Context Authentication Using Constrained Channels," tech. report HPL–2001–84, Hewlett-Packard Laboratories, Palo Alto, Calif., 2001.

60. P. Eronen and P. Nikander, "Decentralized Jini Security," *Proc. Network and Distributed System Security 2001* (NDSS 2001), The Internet Soc., Reston, Va., 2001, pp. 161–172.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.

the AUTHORS

Tim Kindberg is a senior researcher at Hewlett-Packard Labs, Palo Alto, where he works on nomadic computing systems as part of the Cooltown program. His research interests include ubiquitous computing systems, distributed systems, and human factors. He has a BA in mathematics from the University of Cambridge and a PhD in computer science from the University of Westminster. He is coauthor of *Distributed Systems: Concepts & Design* (Addison-Wesley, 2001). Contact him at Mobile Systems and Services Lab, Hewlett-Packard Labs, 1501 Page Mill Rd., MS 1138, Palo Alto, CA 94304-1126; timothy@hpl.hp.com; www.champignon.net/TimKindberg.

Armando Fox is an assistant professor at Stanford University. His research interests include the design of robust software infrastructure for Internet services and ubiquitous computing, and user interface issues related to mobile and ubiquitous computing. He received a BS in electrical engineering from the Massachusetts Institute of Technology, an MS in electrical engineering from the University of Illinois, and a PhD in computer science from University of California, Berkeley, as a researcher in the Daedalus wireless and mobile computing project. He is an ACM member and a founder of ProxiNet (now a division of PumaTech), which is commercializing thin client mobile computing technology developed at UC Berkeley. Contact him at 446 Gates Bldg., 4-A, Stanford Univ., Stanford, CA 94305-9040; fox@cs.stanford.edu; http://swig.stanford. edu/~fox.