Standards & Emerging Technologies

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

Programming Pervasive Spaces

Sumi Helal

Like it or not, tens of billions of lines of Cobol code are still in use today. Invented in 1959 by a group of computer professionals, Cobol empowered developers worldwide to program the mainframe and create applications still in existence today. Undoubtedly, Cobol owes much of its success to its standardization, which started with the American National Standard in 1968.

Yet these days, standards alone won't lead to success. With the invention of the PC and emergence of the network, we realized we also need new concepts and capabilities to program networks of computers. Standards such as TCP/IP and IEEE 802 played a major role in transforming the first computer network concept (Arpanet) to the Internet we know today. However, we also had to invent new computing models such as the client-server model, transactions, distributed objects, Web services, disconnected operation, and computing grids. Furthermore, we had to invent various middleware to support these emerging models, hiding the underlying system's complexity and presenting a more programmable view to software and application developers.

Today, with the advent of sensor networks and pinhead-size computers, we're moving much closer to realizing the vision of ubiquitous and pervasive computing. However, as we create pervasive spaces, we must think ahead to consider how we'll program them—just as we successfully programmed the mainframe and, later on, the Internet.

INTEGRATED ENVIRONMENTS AND THEIR LIMITATIONS

Researchers have recently developed various pervasive computing systems and prototypes to demonstrate how this new paradigm benefits specific application domains (such as homeland security, successful aging, entertainment, and education). In most cases, the researchers followed a system integration approach, interconnecting various physical elements and devices including sensors, actuators, microcontrollers, computers, and appliances using several networks and connectors. Unfortunately, many of these systems and prototypes lack scalability and are closed to third parties. Furthermore, they have yet to demonstrate their ability to evolve as new technologies emerge and as our understanding of a specific application area matures.

Nonscalable integration

Any pervasive system is bound to consist of numerous heterogeneous elements that require integration. Unfortunately, the system integration task itself, which is mostly manual and ad hoc, usually lacks scalability. There's a learning curve associated with every element in terms of first understanding its characteristics and operations and then determining how best to configure and integrate it. Also, every time you insert a new element into the space, there's the possibility of conflicts or uncertain behavior in the overall system. Thus, tedious, repeated testing is required, which further slows the integration process.

Consider a temperature sensor that needs to be connected to an embedded Java program to periodically report a refrigerated truck's temperature. Say you have two boxes—one from Hewlett-Packard containing an iPAQ running Linux, and another from Maxim containing a one-wire temperature sensor. These will require quite a bit of physical and hardware interfacing. In addition, you'll need to write low-level software to interact with the sensor. Even worse, change the sensor, vendor, or PDA operating system and you might have to completely rewrite the software.

Closed-world assumptions

Another problem is that an integrated environment is a relatively closed system—it's not particularly open to extensions or expansion, except perhaps accidentally. It's tightly coupled to a combination of technologies that happened to be available at development time. It's thus difficult—if not impossible—to add new technologies, sensors, and devices after system integration is complete.

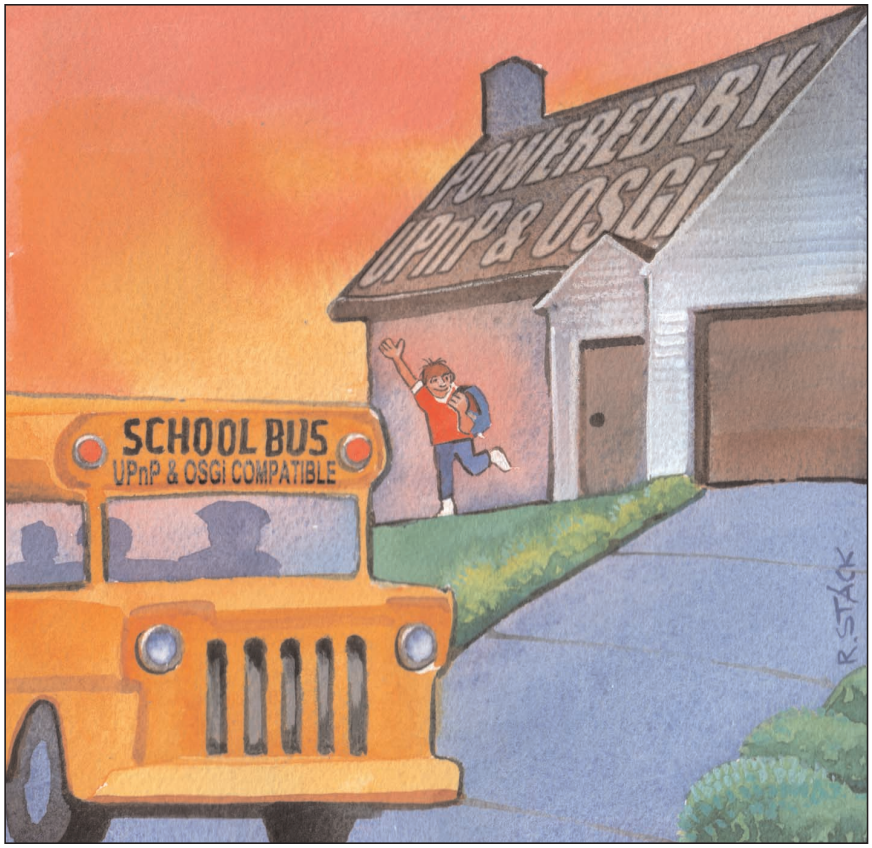
An integrated environment is also closed and restricted to only a few participants—the designers, system integrators, and users. There's no easy way to let a third party participate. For example, an energy- and utility-efficient smart home developed in 2005 might not be compatible with a utility-saving sprinkler system developed in 2006 by a third-party vendor.

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

Fixed-point concepts

Also problematic is the fact that our experience in building integrated environments is limited by the set of concepts we know at the time of development. This might sound like an always-true statement regardless of whether we're doing pervasive, mobile, or distributed computing, but it's especially troubling in pervasive computing.

Take smart homes, for example. Unlike Nokia phones, you can't upgrade and replace them every six months. Once built for a specific goal (such as to assist the elderly or handicapped, save power, or support proactive health for an entire family), the home will likely be used for decades to come. That's why we need to ensure that our smart spaces will be compatible with not-yet-developed concepts. This might not be realistic, but smart pervasive spaces are bound to outlast any known set of pervasive computing concepts. Service gateways and context awareness are two examples of recent concepts that have steeply influenced how we think of pervasive computing. Surely other new concepts are on the horizon.



THE NEED FOR MIDDLEWARE AND STANDARDS

Moving beyond integrated environments will require a middleware that can automate integration tasks and ensure the pervasive space's openness and extensibility. The middleware must also enable programmers to develop applications dynamically without having to interact with the physical world of sensors, actuators, and devices. In other words, we need a middleware that can decouple programming and application development from physical-space construction and integration.

Self-integration

Universal Plug and Play and other service discovery protocols are difficult to ignore when considering a middleware for pervasive computing. UPnP lets home computer owners connect devices to their PCs without having to manually integrate them (that is, without having to install drivers, for example). We similarly need a middleware that lets elements in a pervasive space integrate themselves automatically into that space. Such self-integration would lead to scalable, economical, and open pervasive computing—scalable and economical because we'd no longer need human system integrators (engineers or technicians working hundreds of hours and charging thousands of dollars), and open because third parties implementing sensors or devices could participate at any time in the pervasive space's life cycle.

Self-integration, however, requires a standard—which could be based on UPnP, OSGi, or other existing or new standards. The challenge is to find a standard whose adoption is possible by a broad category of vendors ranging from appliances to consumer electronics to electric and electronic boards and components. It should be just as easy for, say, a heat sensor manufacturer to implement the standard as it would be for a plasma display manufacturer. If both the heat sensor and the plasma display were equally able to advertise their presence and register their services once brought into a space, we'd be much better off than we are now with integrated environments.

In reality, however, many sensors and other elements in a pervasive space can't participate in any standard or nonstandard protocols. A heat sensor, for instance, doesn't have any processing or memory capabilities to engage in any protocols. How can such a sensor be self-integrated? It can't, at least not without a *sensor platform*—a hardware middleware that the (heat) sensor manufacturers supply. Sensor platforms don't have to be powerful computers. They only need to carry on board their sensors' service definitions. For instance, a sensor platform might only need an embedded microcontroller and a small EEPROM memory storing UPnP XML data, or an OSGi Bundle or its URL.

Appliances are another challenge to the concept of self-integration. How can you integrate a floor lamp into a space? Again, you can't, unless you have another piece of middleware.

STANDARDS & EMERGING TECHNOLOGIES

Given that most appliances use power plugs, perhaps we could invent a "smart plug" hardware middleware that could integrate floor lamps, irons, microwaves, and the whole world of appliances into smart spaces.

Unless the pervasive computing research community pays more attention to middleware and its value, we'll continue to waste and duplicate our efforts. Fortunately, Smart-Its is one prototyped concept that's a step in the right direction (for more information, see "Physical Prototyping with Smart-Its," *IEEE Pervasive Computing*, July–Sept. 2004, pp. 74–82). Another strong contribution to the middleware movement is Smart-Plugs (see www.harris.cise.ufl.edu), which the University of Florida is currently prototyping. It's also encouraging to see a glimpse of such middleware available today as commercial products: Phidgets (or Physical Widgets) cater to a limited extent to this middleware thinking (see www.phidgetsusa.com).

Self-integration in pervasive computing thus seems feasible and within reach. We just need to find a sensible framework to define it and then select a widely accepted standard. Anyone volunteering to take on these tasks?

Semantic exploitation

Any middleware we use should also extend self-integration to include service semantics in addition to the service definition so that a joining entity could explore and fully participate in the space. For example, the temperature sensor (via its sensor platform) could offer information about its domain values (such as whether it measures in Celsius or Fahrenheit). It could also suggest other aggregated services that it could offer if and when other services become available in the space (such as a climate sensor if a humidity sensor is added). Exploiting semantics will let the pervasive space's functionality and behavior develop and evolve.

Space-specific ontologies will enable such exploitation of knowledge and semantics in pervasive computing. This again seems feasible and within reach. Ontologies for smart homes have started to emerge, so it shouldn't be too difficult to define other important ontologies such as for a classroom, coffee shop, bus station, bus, train, or airport terminal, to mention just a few.

PROGRAMMABILITY

A critical goal for middleware is to present application developers with a programmable environment. In other words, the middleware should create and activate the functionality of an otherwise self-integrated yet application-less pervasive environment. If the middleware fulfills this role, it'll create a new paradigm in which the process of creating and integrating the physical world is separate from the process of designing and "engineering" the specific desired applications. By comparison, an integrated environment has the applications integrated and bundled with everything else.

The middleware should let programmers perceive the smart space as a runtime environment and as a space-specific software library for use within a high-level language. For example, it should present all sensors and actuators in a form ready for use—perhaps as a *service*. With special support to browse and learn such a dynamic library of services, a programmer should be able to immediately use the space sensors and actuators from within the application. Service composition would then be a natural model for developing applications on top of this middleware.

Having a service view of every sensor and actuator will enable rapid prototyping and a much faster development life cycle. For example, suppose you want an application that can control ambient light when the TV is on. A programmer quickly browses the space and identifies a room-light sensor service, a window-blind sensor service, and a TV actuator service. The programmer could then easily develop logic that uses all these services to determine a possible action, which could in turn use additional services. So, an action might use the light-dimming actuator service and possibly the motorized-blind actuator service to bring light to ambient conditions.

The middleware's programmability aspect will not only empower application development but also support the notion of context-aware application development. Assuming a simple definition of *context*—"a particular combination of sensor states"—it should be straightforward for programmers to define contexts as special-service compositions of relevant sensor services. Programmers could vary context sensors' properties to allow a variety of context production and consumption models.

Who should program a pervasive space?

It shouldn't be surprising if computer and IT professionals act as pervasive computing programmers. Using middleware and standards, they should be productive and focus on the application's goals. It should be easy to train such programmers and thus to create a whole developer community. However, we can't gain the benefits of pervasive computing without involving domain experts in application development. A psychiatrist, for instance, would be the best individual to program an at-home application to detect if insomnia is an experimental treatment's side effect. Similarly, a gastroenterologist would be the most appropriate person to test, nonintrusively, if an elderly patient's peptic ulcer is the result of an H. pylori bacteria caused by insufficient hygiene. Envisioning such scenarios helps reveal the need to change our programmer model to accommodate domain experts as well as computer professionals.

Programming models

Object discovery, reflection, and brokerage, all of which can deal with dynamic environments, have been useful mechanisms in object-oriented programming. Yet service discovery and lookup services have proven to be even more effective in such environments. We

need similar concepts to effectively program pervasive spaces. *Space reflection* will be essential in providing application developers with a programming scope. As pointed out earlier, self-integration could provide all the information needed for space reflection in the simple form of service advertisements. Therefore, service-oriented programming seems more appropriate than the object model for pervasive computing. Service-oriented programming is also a much simpler model, which increases the chances of success in broadening the programmer model. Of course, there's lots of room for debate on this issue.

Indeed, we need a programming model that can deliver simple but powerful abstractions for a broad category of programmers. Some context-aware programming toolkits appear to fill this need.[1] A programmer presented with a *context/condition/action* model (similar to the event/condition/action model followed in active databases and other systems) might quickly develop better applications, but we need to further investigate this.

To achieve dependability and cope with behavior uncertainty, we need some "global" programming. Such programming won't be application-specific and could be equivalent to administrative programming of distributed system tools and monitors. Any proposed programming model should address this need.

Integrated Development Environments?

IDEs have revolutionized programmers' productivity and promoted the adoption of good software engineering practices. There's an even greater need for IDEs in pervasive computing. (Don't confuse IDE with IE—integrated environments or, as I call them, first-generation pervasive computing systems). Visual Studio, Forte, and Eclipse are great examples of IDEs.

Imagine a smart space being represented as an Eclipse-like project showing tabs for existing sensors, actuators, and contexts as well as service and context composition tools. Such an IDE would be initialized by pointing it to a space URL instead of to a user workspace file directory. The IDE would use space reflection to initialize and to provide the programmer with the view and scope necessary for application development and debugging. The IDE could even be a graphical development environment in which services, context, and applications are represented or created graphically using a LabView-style building-block interface (see www.ni.com/labview). Developed applications might be committed back to the space as registered services. Without expanding on the range of capabilities these IDEs should offer, it should be obvious that such IDEs will contribute significantly to programming pervasive spaces. We need to get busy prototyping this concept and validating its impact.

My goal here wasn't to suggest any specific directions or endorse any specific standard but rather to stimulate a discussion on the development of pervasive spaces. I welcome your input and invite experts working in this area to share their work and views with this column's readers.

REFERENCE

1. A. Dey, D. Salber, and G. Abowd, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," *Human Computer Interaction J.*, vol. 16, 2001, pp. 97–166.

Sumi Helal is a professor at the University of Florida and is the director of its Harris Mobile and Pervasive Computing Laboratory. He is also president and CEO of Phoneomena, Inc. Contact him at helal@cise.ufl.edu.