One.world: Experiences with a Pervasive Computing Architecture

A new architecture, one.world, provides an integrated, comprehensive framework for building pervasive applications. It targets applications that automatically adapt to highly dynamic computing environments, and it includes services that make it easier for developers to manage constant change.

Robert Grimm
New York University

Pervasive computing provides an attractive vision for accessing information anywhere and anytime.[1] However, a considerable obstacle to realizing this vision is the development of applications that continually adapt to an ever-changing computing environment and still function when people move through the physical world or switch devices. In the *one.world* project,[2] we've tried to create the appropriate system support so that developers can effectively build and deploy adaptable applications. We designed our architecture from the ground up to address the needs of pervasive applications. This architecture includes services that simplify the task of coping with constant change. Notably, *discovery* helps developers build applications that locate and connect to services on other devices, and *migration* helps with applications that follow users as they move through the physical world.

To better understand pervasive computing's potential and challenges, consider researchers working in a biology laboratory. Their goal is to perform reproducible experiments. Yet today they still manually log individual steps in their paper notebooks, easily leading to incomplete experimental records. The use of notebooks also complicates the sharing of results with other researchers unless biologists explicitly enter the data in their PCs. In contrast, a digital laboratory employs radio frequency identification (RFID) and barcode scanners to easily capture data, location sensors to track researchers' movements in the laboratory, and touch screens to display experimental data close to the researchers. Thus, biologists working in a digital laboratory have more complete records of their experiments and can more easily share results with their colleagues.

However, implementing the digital biology laboratory with contemporary system services is difficult because such services typically assume a relatively static and well-administered computing environment. Furthermore, technologies such as remote procedure call (RPC) and distributed objects tend to hide distribution from applications, so if changes occur, people must manually adapt the system instead of applications adapting for them. For example, with contemporary systems, it's hard for researchers to move between devices in the digital laboratory. Every time, they must manually log in, start their applications, and load their documents. Similarly, it's hard to integrate visiting researchers' devices because people must first configure them—for example, to use the correct wireless network. Finally, it's difficult to share experimental data because researchers must explicitly manage shared files and convert between different formats.

Published by the IEEE CS and IEEE ComSoc ■ 1536-1268/04/$20.00 © 2004 IEEE

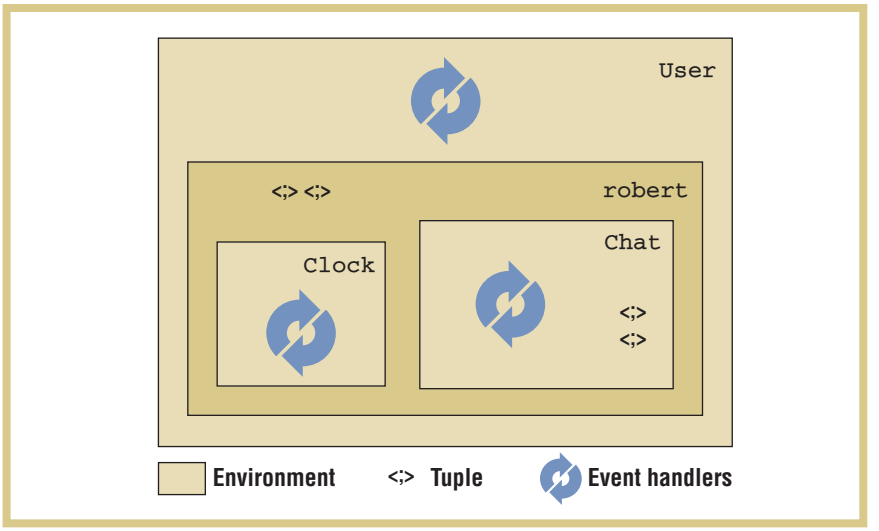


Figure 1. Example environment hierarchy. The User environment hosts Emcee, *one.world*'s user and application management utility. User has one child, robert, which stores several tuples representing that user's preferences. The robert environment also has two children: Clock contains only active event handlers; Chat hosts the instant-messaging application and stores tuples representing the music that this application broadcasts.

Overview of *one.world*

To overcome the limitations of contemporary distributed systems, we've identified three requirements for system support for pervasive applications. First, as people move through the physical world, either carrying their own portable devices or switching between devices, an application's location and execution context constantly changes. So, system support must embrace contextual change, not hide it from applications. Second, users expect that their devices and applications will just plug together. Thus, system support must encourage ad hoc composition and not assume a static computing environment with just a few interactions. Third, as users collaborate, they must easily share information. Hence, system support must facilitate sharing between applications and between devices.

Our architecture is centered around meeting these three requirements. It employs a classic user/kernel split: Foundation and system services run in the kernel; libraries, system utilities, and applications run in user space. *One.world*'s foundation services directly address these individual requirements. They also provide the basis for our architecture's system services, which in turn serve as common building blocks for pervasive applications.

The four foundation services are a *virtual machine*, *tuples*, *asynchronous events*, and *environments*. First, all code in *one.world* runs in a virtual machine—namely, the Java virtual machine (JVM). Because of pervasive computing environments' inherent heterogeneity, developers can't possibly predict all the devices that their applications will run on, so the JVM ensures that applications and devices are composable. Second, *one.world* represents all data as tuples, which define a common data model, including a type system, for all applications and thus simplify data sharing. Tuples are records with named and optionally typed fields. Moreover, each tuple is self-describing, so an application can dynamically inspect its structure and contents. Third, *one.world* expresses all communication, whether local or remote, through asynchronous events. These events serve to explicitly notify applications of changes in their runtime context.

Finally, environments are *one.world*'s main structuring mechanism. Like traditional operating system processes, environments host running applications and isolate them from one another. They also serve as containers for persistent data, providing associative tuple storage and thus making it possible to group running applications with their persistent data. Furthermore, environments nest within one another, making it easy to extend and compose applications. An outer environment has complete control over all nested environments, including the ability to easily intercept and modify events sent by inner environments to *one.world*'s kernel (which runs in a device's root environment) and to other devices. This interposition facility lets developers and users dynamically change an application's behavior without changing the application itself. Moreover, it's particularly useful for complex and reusable behaviors, such as replicating an application's data or deciding when to migrate an application. Figure 1 shows an example environment hierarchy.

One.world's system services build on the foundation services and serve as common building blocks for pervasive applications. Table 1 summarizes spe-

TABLE 1
Application needs and *one.world*'s corresponding system services.

Application need	*One.world* service
Search	Query engine
Store data	Structured I/O
Communicate	Remote events
Locate	Discovery
Fault-protect	Check-pointing
Move	Migration

cific application needs and *one.world*'s corresponding system services. Of these services, discovery and migration are probably the most interesting.

Discovery locates resources—that is, event handlers—by their descriptions. It leverages *one.world*'s uniform data model, in which all data, including events and queries, are tuples. Using this data model, discovery supports a rich set of options, including early and late binding, anycast, and multicast, with only three simple operations. Furthermore, because discovery is an essential service—without it, applications can't adapt to a new or changing runtime context—it's also self-managing. One device acts as the discovery server, providing the mapping between descriptions and event handlers. All devices running *one.world* that are visible on the local broadcast network automatically elect the server from among themselves. To ensure availability, *one.world* calls elections aggressively, and these elections complete after a fixed time period. The individual devices tolerate any resulting inconsistencies by exporting their discoverable resources to all visible servers while looking up resources on only one server.

Migration moves or copies an environment and all its contents to a different device, thus simplifying the implementation of applications that follow a person through the physical world. Unlike traditional process migration,[3] *one.world*'s migration isn't transparent, and the migrated application's state is limited to the environments being migrated. During migration, *one.world* automatically nulls out references to resources outside the environment tree. This practice is acceptable because applications already expect change. Furthermore, *one.world*'s migration is eager in that it moves the entire state between the devices in one atomic operation. This avoids residual dependencies and requires connectivity between the devices only during migration. The result is that *one.world*'s migration avoids many of the complexities of traditional process migration, and migration across the Internet becomes practical.

In evaluating *one.world*, we tried to determine whether focusing on the three requirements has led to a practical architecture for pervasive applications.

We specifically designed *one.world* to help developers build applications that automatically adapt to an ever-changing computing environment. Discovery helps in locating and connecting to services on other devices, and migration helps in implementing applications that follow a user through the physical world. The advantage of our architecture's services over other services is that they're built from the ground up to embrace change, encourage ad hoc composition, and facilitate sharing.

Evaluation criteria

In evaluating *one.world*, we tried to determine whether focusing on the three requirements has led to a practical architecture for pervasive applications. However, because this question is rather general and difficult to answer, we relied on four more specific criteria and corresponding questions:

- *Completeness*. Can we build useful programs using *one.world*'s primitives? This criterion determines whether our architecture is sufficiently powerful and extensible to support interesting user-space programs, including additional services and utilities akin to the Unix shell.
- *Complexity*. How hard is it to write code in *one.world*? This criterion determines the effort involved in developing programs for our architecture. We're especially interested in how making applications adaptable impacts programmer productivity.
- *Performance*. Is system performance acceptable? This criterion determines whether our architecture performs well enough to support actual application workloads. Because our goal is to make applications adaptable, we're especially interested in whether applications respond quickly to changes in their runtime context.
- *Utility*. Have we fostered success for others? This criterion determines whether others can build real pervasive applications atop *one.world*. It also represents the most important criterion. After all, a system architecture is only as useful as the programs running on top of it.

Services and applications atop *one.world*.

To answer these questions, we and others have built several services, utilities, and applications atop *one.world*. In particular, we've built a replication service, a user and application manager called Emcee, and a text-and-audio-messaging system called Chat. We've also supported the University of Washington's Labscape project in porting their digital biology laboratory assistant to *one.world*.[4] The Labscape team deployed this laboratory assistant as part of the Cell Systems Initiative. Besides these primary programs, students have built, atop our architecture, a music-sharing system; a messaging system for future, intelligent home appli-

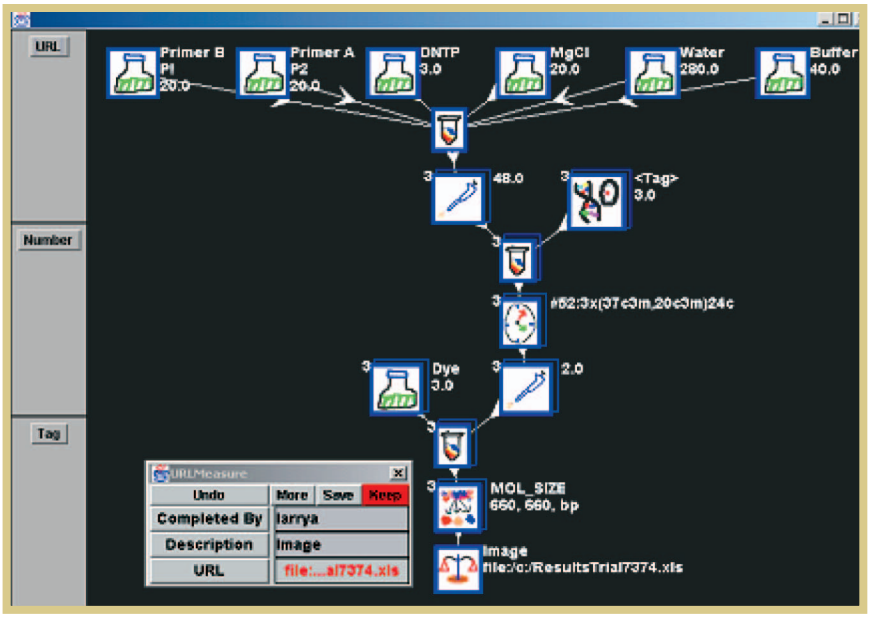


Figure 2. Labscape's user interface. An experimental flowgraph, or guide, represents each experiment.

ances; a graphical debugger; and a Web server. We now summarize our own applications and the Labscape digital biology laboratory assistant, and also the corresponding experimental results; an earlier work provides more details.[2]

Replication service

To provide ubiquitous access to information, pervasive applications must access the corresponding data items, even if several people share the same data and access it from different, possibly disconnected devices. Our replication service helps address this need. We patterned this service after Jim Gray and his colleagues' two-tier replication model,[5] in which a master owns all data, and replicas host copies of that data. The replication service runs in user space and exploits *one.world*'s environment nesting to interpose on an application's access to tuple storage. In connected mode, updates are final, and the system forwards them directly to the master. In disconnected mode, updates are tentative, and the system logs them on the replica in a separate log environment. When a replica connects again, it synchronizes with the master by replaying its log against the master and receiving any updates from the master. Synchronization is implemented by migrating a copy of the log environment to the master, where an application-specific component replays the updates and, if necessary, performs conflict resolution.

User and application manager

Emcee includes support for setting up users, running applications for a user, and check-pointing all of a user's applications. Emcee also lets users move or copy applications and their data through a simple drag-and-drop interface, or move all their applications between devices. Users can either push applications from the current device to another one or pull them from another device to the current one. As Figure 1 shows, the implementation structures the environment hierarchy according to the pattern */User/user/application* and directly builds on this environment nesting to control users' applications. The implementation relies on check-pointing and migration to save, restore, move, and copy applications, and on discovery to locate a user's applications when pulling them to a new device.

Text-and-audio-messaging system

Chat is based on a simple model under which users send text and audio messages to a channel and subscribe to that channel to see and hear the messages sent to it. The implementation sends all messages through late-binding discovery, which automatically routes messages to all interested parties by combining the lookup of matching event handlers with the actual event delivery; subscribing to a channel simply translates to exporting an event handler to discovery. Therefore, Chat automatically adapts to channel subscription and subscriber location changes. Users can stream audio either from a microphone or from sound tuples stored in an environment. *One.world* provides utilities that let users import sound files from a traditional file system and convert them to a sequence of sound tuples.

Labscape

A fundamental feature of the Labscape application is that experimental data follows researchers as they move through the laboratory. At the same time, there's no need to move the entire application, only a small component to capture and display experimental data. Eventually, Labscape forwards all data to a central repository—making it possible, for example, to mine the data of several experiments. Figure 2 shows a snapshot of the capture-and-display component's user interface, which is called a *guide*. The Labscape team developed this interface through user interface studies with biology researchers. The individual icons represent different experimental steps, and the arrows represent ordering constraints. Initially, the guide functions as a plan for performing the experiment. As researchers perform a step, they annotate the corresponding icon with the results of that step. Thus, the guide eventually becomes a record of the experiment.

Labscape's implementation relies on several services, which all communicate through *one.world*'s late-binding discovery and, consequently, are isolated from other services' locations and from transient service and network failures. The *device access service* collects exper-

imental data from RFID and barcode scanners, and collects location updates from infrared sensors. This service converts the data and the updates into appropriate *one.world* events and then forwards them to the *proximity service*, which tracks researchers' locations. For experimental data, the proximity service determines which researcher performed the scanning operation and then forwards the data to the researcher's guide. For location updates, the proximity service updates its internal data structures and then directs the researcher's guide to move to the closest touch screen, using *one.world*'s migration. The *state service* is the final repository for all experimental data, which it receives from the researchers' guides. The *WebDAV service* then publishes this data on the Web.

Experimental results

Having described the main programs built atop our architecture, we now return to the four criteria and present the results of our experimental evaluation.

Completeness

The programs just discussed clearly show that *one.world* is powerful enough to support a variety of useful programs. Furthermore, our replication service and Emcee demonstrate that it's possible to implement services and utilities in user space. The key feature for enabling user-space services and utilities is *one.world*'s environment hierarchy, which makes it easy to control other programs and interpose on their event streams.

Complexity

To evaluate the effort involved in writing adaptable applications, we carefully tracked the process of implementing Emcee and Chat. Overall, implementing the two programs took 256 hours for 4,321 noncommenting source statements (NCSS), yielding a measured productivity of 16.5 NCSS/hr. This productivity is within the range of results reported in the literature,[6,7] suggesting that writing adaptable applications is not significantly harder than writing more conventional applications.

Our implementation of discovery performs well enough to route several independent streams of uncompressed audio data.

Performance

We conducted measurements with Emcee and Chat using Sun's Java HotSpot Virtual Machine 1.3.1 running on Dell 800-MHz Pentium III PCs connected through a 100-Mbyte switched Ethernet. These performance measurements show that service interruptions due to migration or discovery server elections last less than 4 seconds, comparing favorably, for example, with the Transmission Control Protocol (TCP) timing out after several minutes. Furthermore, although migration latency generally depends on the number and size of stored tuples, this latency is only 7 seconds for an environment storing 8 Mbytes of audio data, which is fast enough for a person moving through the physical world. Finally, our implementation of discovery performs well enough to route several independent streams of uncompressed audio data. However, its scalability is also limited, supporting no more than 10 streams for the PCs on our network.

Utility

The Labscape digital laboratory assistant illustrates the utility of *one.world* as a real-world application substrate. Furthermore, in contrast to an earlier version, which the Labscape team implemented using Java sockets and the team's own application-specific migration instead of the corresponding *one.world* services, the *one.world* port required less than half the development time, had an order-of-magnitude faster migration time (seconds instead of minutes), had a two-orders-of-magnitude longer mean time between failures (days instead of minutes), and could recover piecemeal from failures rather than requiring a restart of the entire lab every time. We attribute these improvements to two main factors. First, our architecture's services—including the level of indirection offered by discovery rather than direct TCP connections—provide a better match to the Labscape application's needs while also exposing a cleaner API. Second, by using *one.world*, Labscape's developers could focus on the application itself rather than having to create necessary system support.

Experiences and lessons learned

On the basis of the results just discussed, we conclude that *one.world* does, in fact, let developers effectively build applications that adapt to change. Moreover, we conclude that our focus on embracing change, encouraging composition, and facilitating sharing has led to a practical architecture for pervasive applications.

Nevertheless, the user-space programs we and others have built have not only provided a solid basis for evaluating *one.world*, they've also helped us gain a better understanding of our architecture's strengths and limitations. We now focus on the resulting insights and identify lessons that are applicable beyond our work.

Clear successes

The central role played by environ-

ments in our architecture implies a more general pattern—namely, that nesting is a powerful paradigm for controlling and composing applications. Nesting provides control, as both Emcee and the graphical debugger illustrate. Moreover, developers and users can use nesting to extend applications, as our replication service illustrates. Nesting thus makes it possible to easily factor important, possibly complex behaviors and provide them as common application building blocks. Furthermore, nesting is attractive because it preserves the relationships among environments. For example, when we stored audio tuples in a child environment of Chat's environment, the former remained a child, even if Chat's environment moved between devices. This nesting flexibility is also present in the Ambient Calculus,[8] which, similar to *one.world*'s environments, groups computations and data in nested containers.

The use of migration in Emcee and Labscape illustrates that migration provides a general building block for structuring pervasive systems. Migration can be an internal tool, as with our replication service, where it replaces a networked reconciliation protocol. It can also be an application-level tool—either controlled from the outside (as with Emcee) or self-initiated (as with Labscape's guides). Furthermore, migration can be used simultaneously as an internal and application-level building block. For example, our replication service's master and its replicas are migratable. Migrating the master is useful when, for example, upgrading the computer on which the master runs; migrating a replica is useful when the user is switching devices. However, migration's widespread use also requires support for discovery so that migrating services and applications can reconnect after a location change.

Besides representing a central application building block, *one.world*'s migration service also leverages our architecture's other services as much as possible to avoid complexity and to provide a clean model for its operation. For example, it relies on the JVM to provide a uniform execution environment across different devices and hardware architectures. It also relies on environments to clearly delineate which state to move between devices and which state not to move. Furthermore, the migration service relies on asynchronous events to capture an application's execution state in the form of its event queues and to notify the application of a completed move or copy operation. More fundamentally, however, *one.world*'s migration can avoid many of the complexities associated with providing such a service because applications already expect change.[3] In other words, exposing distribution enables not only more adaptable applications but also more powerful system services.

The central role played by environments in our architecture implies a more general pattern—namely, that nesting is a powerful paradigm for controlling and composing applications.

Need for user-space support

Consistent with our goal of exposing change, all communications in *one.world*, whether local or remote, occur through asynchronous events. Furthermore, event delivery has at-most-once semantics. For remote communications, at-most-once semantics are appropriate because, in lieu of transactional delivery protocols, a remote device might fail after accepting an event but before delivering it to the intended application. For local event handling, exactly-once delivery is the norm. However, at-most-once semantics let *one.world*'s implementation recover from pathological overload conditions by selectively shedding load.

Although we still believe that best-effort, asynchronous events are the appropriate kernel-level services for pervasive computing, many applications rely on synchronous request-and-response interactions. This raises the question of how to best implement these interactions—a concern Labscape's developers specifically noted. After some experimentation, we found the following approach, which we call the *logic/operation pattern*, particularly successful. Under this pattern, an application is partitioned into logic and operations, which are implemented by separate sets of event handlers. Computations that don't fail are logic; interactions that may fail—notably all I/O—are operations. A user-level *operation library* simplifies the implementation of event handlers representing operations by maintaining the state associated with request-and-response interactions and by detecting and recovering from failures through timeouts and retries. The operation library conveys a failure condition to the appropriate logic only if recovery fails repeatedly or a failure cannot be recovered from in a general way. Although the operation library can't hide *one.world*'s event-based programming model from developers, it does enable a more familiar synchronous programming style and reduces the complexity of building event-based applications.

To enforce protection, our architecture prevents applications from accessing Java's `java.lang.System` class, and it makes select methods (notably `arraycopy()` to copy the contents of arrays and `getProperty()` to

access system properties) accessible through its own SystemUtilities class. Using a different class to access these methods doesn't restrict applications written from scratch; developers can simply use a different class name in the source code. However, it does prevent existing Java libraries, which frequently employ these methods, from running on *one.world*. As a result, the Labscape team had difficulties reusing third-party libraries. To address this issue, we developed a simple utility that, using binary rewriting, transforms existing libraries and replaces invocations to System's methods with the corresponding *one.world* methods.

The larger lesson behind both our operation library and the binary rewriting tool is that, to be effective, a system architecture should provide a programming environment as close as possible to familiar development platforms. Otherwise, application developers will focus on working around the system architecture's peculiarities rather than focusing on their applications.

Overdesigned features

When designing *one.world*'s interfaces, we took a cue from Unix and designed structured I/O to expose the same basic interface for both storage and communications (although they're distinct services, just like files and sockets are distinct services in Unix). However, none of the programs we and others have built use structured I/O networking; they all rely on remote events and discovery for networked communications. Only the implementation of remote events and discovery in the kernel builds on structured I/O networking. Developers tend to favor remote events and discovery because they're higher-level, more flexible services. So, we believe we overdesigned structured I/O. We could have omitted structured I/O networking and instead used a simpler, internal networking layer for implementing remote events and discovery. In other words, storage and communications are orthogonal to each other and best implemented by separate services with distinct interfaces.

Defining an appropriate data-centric data model is an important topic for future research. The challenge is to define a data model that meets conflicting requirements.

Because structured I/O storage provides a record-based interface to persistent storage, we also took a cue from conventional databases and provided atomicity, isolation, and durability for all operations. However, our performance evaluation of the replication service suggests that the durability guarantees can generate excessive overhead for some applications. In particular, immediately forcing each write operation to disk is unnecessary when logging updates in disconnected mode because all updates are already tentative. To address this problem, we designed (but have not implemented) a simple extension to structured I/O, under which applications can request that the destructive write and delete operations provide only relaxed durability and are lazily written to disk. Just as with traditional file systems, applications using this option must explicitly perform a flush to force pending updates to disk. When we consider our experience with structured I/O contrasted with operations to provide reliable event delivery, we conclude that, just as with other end-to-end guarantees, it's best to provide reliability guarantees in user space rather than as the default within the kernel. That way, applications that don't need such guarantees need not pay the overhead.

Limitations

The biggest limitation of our architecture is that like Jini[9] and Lime[10] but unlike iROS[11] and L^{2}imbo,[12] *one.world* implements tuples through Java classes, using public fields to represent a tuple's values. Implementing tuples through Java classes provides a convenient interface to data for applications because accessing a value is a simple field access. However, it also poses a considerable problem for services such as discovery, which process many different types of data for many different applications and must access the corresponding class files, in addition to the actual tuples. The fundamental problem is that we've taken a single-node programming methodology—a programmatic data model—which expresses data schemas in the form of code, and applied it to a distributed system. This suggests that we need to abandon the programmatic data model altogether and instead use a data-centric data model, such as XML Schema, which expresses schemas as data rather than code. With a data-centric data model, applications and services must still access a data item's schema to manipulate the data item. However, because the schemas are data themselves and not code, they're easier to distribute and share, and they aren't tied to a specific execution platform. Hence, data-centric data models provide better interoperability than programmatic data models.

Defining an appropriate data-centric data model is an important topic for future research. The challenge is to define a data model that meets conflicting requirements. On the one hand, to support the pervasive sharing of information, the data model must be general and sup-

ported by a wide range of platforms. One possible starting point is XML Schema. It already defines the data model for SOAP, the emerging standard for remote communications between Web services that, for example, Microsoft's .NET platform uses. On the other hand, the data model must be easy to program and efficient to use. For an XML-based data model, this means avoiding the complexities of a general data access interface, such as the Document Object Model, and providing a more efficient encoding. Ideally, a data-centric data model should be as easy to program as field access for tuples in our architecture, while also avoiding the need for exchanging class files between devices.

In contrast to iROS, which includes support for HTTP in addition to its own tuple-based protocol, *one.world* exclusively builds on its own networking protocols in the form of remote events and discovery. In our experience, these remote-events and discovery protocols have clearly simplified the implementation of pervasive applications running on our architecture. At the same time, using nonstandard protocols has made it harder to integrate *one.world* with other distributed systems, notably Web-based applications.

Although we've successfully implemented a Web server atop our architecture, integrating outside applications with *one.world* through remote events or discovery is impractical, especially if the outside applications aren't written in Java. Because of our programmatic data model, an outside application would have to reimplement large parts of Java's object serialization, which would be unnecessarily complex. However, to provide ubiquitous information access, pervasive applications must easily interact with one another and with Internet services, independent of the underlying system platform. Moving to a data-centric, XML-based data model and using standardized communication protocols such as HTTP will help provide better interoperability between pervasive applications, even if they run on different system architectures. To put it differently, modern distributed systems must be compatible with Internet protocols first and offer additional capabilities second.

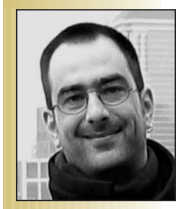
Metrics

Our experimental evaluation of *one.world* certainly represents a first step toward thoroughly evaluating a pervasive computing architecture. But it also illustrates that we need better metrics for designing and evaluating pervasive computing systems. In particular, although we've used NCSS/hr as a productivity metric, we've found only a few references in the literature to calibrate our measurements. Furthermore, the software engineering community seems to have adopted function-point counting as a preferred metric.[13] Function-point counting attempts to capture an application's logical inputs, transformations, and outputs, requiring the development of a higher-level model to measure the software. Thus, function-point counting precludes the automated collection of application statistics, which raises the question of how objective this metric really is. Consequently, we still believe that NCSS/hr provides an appropriate starting point for measuring programmer productivity. However, to determine this metric's accuracy, we need more experience using it across a wider range of applications, system platforms, and programming languages.

We've also characterized *one.world*'s adaptability by following the framework described by Brian Noble and his colleagues and measuring how our architecture's applications and services react to different stimuli.[14] However, we lack standardized benchmarks that can capture the scalability and adaptability requirements of pervasive computing environments. We're especially interested in the number of people and devices that pervasive computing environments must support, the different classes of devices people use, and the corresponding arrival and departure rates of people moving through the physical world. We believe that wireless mobility studies, such as the one performed at Dartmouth University,[15] can provide a good starting point for developing such benchmarks, but there's clearly a need for similar studies to characterize mobility for different organizations and target audiences, and to track changes as mobile devices and wireless technologies develop over the years.

Our experiences with *one.world* suggest several topics for future research into system support for pervasive applications. Notably, defining data models and communication protocols to better integrate pervasive systems with one another and with Internet services remains a challenge. Additionally, metrics for designing and evaluating pervasive systems are still lacking. Our experiences also suggest that, given appropriate system support, pervasive applications are fairly stylized in their implementations, mostly encoding how to route and transform data and where to make their services available. Hence, we envision a different approach to the development of pervasive appli-

the AUTHOR



Robert Grimm is an assistant professor of computer science at New York University. His research interests focus on how to use programming language and compiler technologies to more effectively build distributed systems that scale and evolve more gracefully. Grimm received his PhD in computer science and engineering from the University of Washington. He is a member of the IEEE, the ACM, and Usenix. Contact him at New York University, Dept. of Computer Science, 715 Broadway, Room 711, New York, NY 10003; rgrimm@cs.nyu.edu; www.cs.nyu.edu/rgrimm.

cations—one that favors higher-level, declarative specifications rather than explicitly programming behaviors. For example, such an approach would let developers specify data integrity constraints for replicated storage and policies for migrating pervasive applications. The key insight is that a declarative specification can concisely describe a system's properties, which can then be automatically translated into appropriate actions. In effect, such an approach would treat a pervasive systems platform, such as *one.world*, as the assembly language for implementing complex behaviors. As such, it could significantly simplify the development of complex systems. P

ACKNOWLEDGMENTS

Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Ben Hendrickson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall all contributed to the development of *one.world*. On their behalf, I thank the members of the Labscape project, the students of the University of Washington's CSE 490dp, Daniel Cheah, and Kaustubh Deshmukh for using *one.world* to build their applications. Also, Armando Fox, Ben Goldberg, David Mazières, and the anonymous reviewers provided valuable feedback on earlier versions of this article. For more information on *one.world*, including its source release, see www.cs.nyu.edu/rgrimm/one.world.

REFERENCES

1. M. Weiser, "The Computer for the Twenty-First Century," *Scientific Am.*, vol. 265, no. 3, 1991, pp. 94–104.
2. R. Grimm, *System Support for Pervasive Applications*, doctoral dissertation, Dept. of Computer Science and Eng., Univ. of Washington, 2002.
3. D. Milojičić, F. Douglis, and R. Wheeler, eds., *Mobility: Processes, Computers, and Agents*, Addison-Wesley, 1999.
4. L. Arnstein et al., "Systems Support for Ubiquitous Computing: A Case Study of Two Implementations of Labscape," *Proc. Int'l Conf. Pervasive Computing*, LNCS 2414, Springer-Verlag, 2002, pp. 30–44.
5. J. Gray et al., "The Dangers of Replication and a Solution," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, ACM Press, 1996, pp. 173–182.
6. P. Ferguson et al., *Software Process Improvement Works!*, tech. report CMU/SEI-99-TR-027, Software Eng. Inst., Carnegie Mellon Univ., 1999.
7. L. Prechelt, "An Empirical Comparison of Seven Programming Languages," *Computer*, vol. 33, no. 10, 2000, pp. 23–29.
8. L. Cardelli, "Abstractions for Mobile Computations," *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, LNCS 1603, J. Vitek and C.D. Jensen, eds., Springer-Verlag, pp. 51–94.
9. K. Arnold et al., *The Jini Specification*, Addison-Wesley, 1999.
10. A.L. Murphy, G.P. Picco, and G.-C. Roman, "Lime: A Middleware for Physical and Logical Mobility," *Proc. 21st IEEE Int'l Conf. Distributed Computing Systems* (ICDCS 01), IEEE CS Press, 2001, pp. 524–533.
11. B. Johanson, A. Fox, and T. Winograd, "The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms," *IEEE Pervasive Computing*, vol. 1, no. 2, 2002, pp. 67–74.
12. N. Davies et al., "L^{2}imbo: A Distributed Systems Platform for Mobile Computing," *Mobile Networks and Applications*, vol. 3, no. 2, 1998, pp. 143–156.
13. D. Garmus and D. Herron, *Measuring the Software Process*, Prentice Hall, 1995.
14. B.D. Noble et al., "Agile Application-Aware Adaptation for Mobility," *Proc. Symp. Operating Systems Principles* (SOSP 97), ACM Press, 1997, pp. 276–287.
15. D. Kotz and K. Essien, "Analysis of a Campus-Wide Wireless Network, *Proc. 8th Ann. Int'l Conf. Mobile Computing and Networking* (MobiCom 02), ACM Press, 2002, pp. 107–118.