

38. System Support for Pervasive Applications

Robert Grimm¹ and Brian Bershad²

¹ New York University, New York, NY
rgrimm@cs.nyu.edu

² University of Washington, Seattle, WA
bershad@cs.washington.edu

38.1 Introduction

Pervasive, or ubiquitous, computing [38.4] has the potential to radically transform the way people interact with computers. The key idea behind pervasive computing is to deploy a wide variety of computing devices throughout our living and working spaces. These devices coordinate with each other and network services, with the goal of seamlessly assisting people in completing their tasks. Pervasive computing thus marks a major shift in focus, away from the actual computing technology and towards people and their needs.

To illustrate this shift in focus, consider, for example, researchers working in a biology laboratory. Their goal is to perform reproducible experiments. Yet, today they manually log individual steps in their paper notebooks. This easily leads to incomplete experimental records and makes it unnecessarily hard to share data with other researchers, as the biologists need to explicitly enter the data into their PCs. In contrast, a digital laboratory employs digitized instruments, such as pipettes and incubators, to automatically capture data, location sensors to track researchers' movements, and touchscreens throughout the laboratory to display experimental data close to the researchers. As a result, biologists working in the digital laboratory have more complete records of their experiments and can more easily share results with their colleagues.

For this vision to become a reality, we need system support that directly helps with building, deploying, and using pervasive applications. However, contemporary system services typically assume a static and well-administered computing environment. Furthermore, they tend to hide distribution from applications, with the result that, if changes occur, people need to adapt the system instead of the applications adapting for them. For example, with contemporary systems it is hard to move between machines, as people need to manually log in, start their applications, and load their documents. Similarly, it is hard to integrate new devices, as people need to first configure them, for example, to use the correct wireless network parameters. Furthermore, it is hard to share data, as people need to explicitly manage shared files and convert between different formats.

We argue that, to be successful, system support for pervasive computing needs to address three major requirements. First, as people move throughout the physical world—either carrying their own portable devices or switching between devices—an application's location and execution context changes all the time. As a result, system support needs to *embrace contextual change* and not hide it from applications. Second, users expect that their devices and applications just plug together. System support thus needs to *encourage ad hoc composition* and not assume a static computing environment with

a limited number of interactions. Third, as users collaborate, they need to easily share information. As a result, system support needs to *facilitate sharing* between applications and devices.

We also argue that it is not sufficient to create new system services that address the three requirements and layer them as middleware on top of existing systems. Rather, system support for pervasive computing needs to be designed from the ground up to embrace change, encourage ad hoc composition, and facilitate sharing. Sun's Jini [38.1], for example, provides distributed events and discovery, both of which are useful for building more adaptable applications. However, Jini is layered on top of Java RMI, a distributed object system that is targeted at static and well-administered computing environments. In particular, Java RMI requires a statically configured infrastructure to run name and, for Jini, discovery servers. Furthermore, Java RMI requires that applications are well-behaved, as it relies on synchronous remote invocations, provides no isolation between objects, and closely couples devices to each other (RMI's distributed garbage collection controls an object's lifetime, thus making one device dependent on other devices on the network). So, to avoid the short-comings of existing systems, we need to create the right services from scratch.

38.2 Architecture of *one.world*

To provide the appropriate system support for pervasive applications, we have designed and implemented a new system architecture for pervasive computing [38.3]. Our architecture, called *one.world* and illustrated in Figure 38.1, is centered around meeting the requirements of embracing change, encouraging ad hoc composition, and facilitating sharing. It employs a classic user/kernel split, with foundation and system services running in the kernel, and libraries, system utilities, and applications running in user space. *one.world*'s foundation services directly address the 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 within a virtual machine, such as the Java virtual machine or Microsoft's common language runtime. Because of the inherent heterogeneity of pervasive computing environments, developers cannot possibly predict all devices their applications will run on, and the virtual machine ensures that applications and devices are composable. Second, all data in *one.world* is represented as tuples. Tuples define a common data model, including a type system, for all applications and thus simplify the sharing of data. They are records with named and optionally typed fields, and they are self-describing, so that an application can dynamically inspect a tuple's structure and contents. Third, all communications in *one.world*, whether local or remote, are expressed through asynchronous events. They provide the means for explicitly notifying applications of changes in their runtime context.

Finally, environments are the main structuring mechanism for *one.world*. They represent the units of local computation and, just like traditional operating system processes, isolate applications from each other. Environments also serve as containers for persistent data. In addition to hosting application code, they provide associative tuple storage

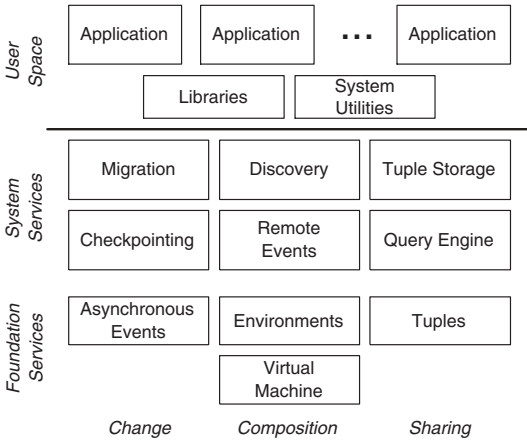


Fig. 38.1. Overview of *one.world*'s architecture.

and thus make it possible to directly group applications with their data. Furthermore, environments can be nested within each other, making it easy to compose applications. An outer environment has complete control over all nested environments, including the ability to interpose on inner environments' communications with *one.world*'s kernel and the outside world. Overall, environments can be thought of as a combination of file system directories and nested processes in other operating systems.

Table 38.1. Application needs and corresponding system services.

Applications need to...	<i>one.world</i> provides...
Search	Query engine
Store data	Tuple storage
Communicate	Remote events
Locate	Discovery
Fault-protect	Checkpointing
Move	Migration

one.world's system services build on the foundation services and serve as common building blocks for pervasive applications. Specific application needs and the corresponding system services are summarized in Table 38.1. Out 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, to support a rich set of options, including early and late binding as well as anycast and multicast, with only three simple operations. More importantly, because discovery is an essential service for pervasive applications—after all, without discovery, applications cannot adapt to a new or changing runtime context—discovery is self-managing and relies on a server that is elected from all devices running *one.world* on the

local network. To ensure availability, elections are called aggressively and complete after a fixed period. The individual devices tolerate any resulting inconsistencies by making their discoverable resources accessible through all 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, *one.world*'s migration is *not* transparent, and migrated application state is limited to the environments being migrated. References to resources outside the environment tree are automatically nulled out during migration, which is acceptable because applications already expect change. As a result, *one.world*'s migration can avoid the complexities of traditional process migration, and migration across the wide area becomes practical.

Overall, *one.world* has been specifically designed to support adaptable applications, with the goal that users do not need to manually adapt their systems. While several services have been explored before, our architecture's services differ in that they have been built from the ground up to embrace change, encourage ad hoc composition, and facilitate sharing.

Experiences. To evaluate *one.world*, we developed several applications, including a text and audio messaging system. We also recruited outside developers. Notably, the Labscape project at the University of Washington ported their electronic laboratory assistant, which works as outlined in the introduction and has been deployed at the Cell Systems Initiative, to *one.world* [38.2]. In contrast to an earlier version, which was implemented using Java sockets and their own application-specific migration, the *one.world* port required less than half the development time, has an order of magnitude faster migration times, two orders of magnitude longer mean time between failures, and can recover piecemeal from failures instead of requiring a restart of the entire lab. While the Labscape team also had some complaints about our event model and the lack of integration with Internet services, the resulting application is good enough to be used by biologists every day and thus establishes *one.world* as solid foundation for building pervasive applications.

38.3 Outlook

We see two major thrusts for future research into system support for pervasive applications, with the first aimed at creating more advanced system services and the second aimed at changing the way we build pervasive applications.

While *one.world*'s system services meet basic application needs, we see three areas for providing additional system services. First, we need better support for reflecting an application's runtime environment to the application, including an ontology for describing device characteristics, network connectivity, and location. For example, *one.world* provides only limited information about a device's capabilities (such as speed and memory capacity) and none about the current level of connectivity. Yet, remaining energy for battery operated devices and cost for network connectivity are important factors when deciding, for example, whether to migrate or to communicate. The main challenge here is to develop an appropriate ontology and the corresponding software sensors.

Second, we need better support for synchronizing and streaming data between devices. While we have experimented with a replication layer that supports disconnected operation and with streaming audio between devices, our current implementations are not sufficiently tuned and not fully integrated with other system services. More importantly, *one.world* does not provide any support for dynamically transforming data, for example, to reduce a video stream's fidelity when sending it across a low bandwidth cellular link. The main challenge here is not how to transform streaming video, but rather how to provide a unified framework for both discrete data (such as experimental data or personal contacts) and streaming data (such as audio or video).

Third, we need better support for changing and upgrading an application's code. *one.world*'s migration can already be used to easily install an application on a new device by simply copying it. However, our architecture does not provide the ability to upgrade applications while they are running. Since many pervasive applications can be expected to be long running, being able to upgrade such applications without disrupting them is an important capability. The main challenge here is to design a mechanism that is automatic, secure, and general enough to also upgrade the system platform itself.

While system support, such as that provided by *one.world*, can simplify the task of developing pervasive applications, developers still need to program all application behaviors by hand. We believe that a higher-level approach is needed to help developers be more effective.

To this end, we turn to the web for inspiration and observe that it uses two related technologies, declarative specifications and scripting, to great effect. For example, web servers typically rely on declarative configuration files that specify how to map a server's virtual name space to actual resources and how to process and filter content. Similarly, web browsers rely on style sheets to specify the appearance of accessed web pages. In both cases, more advanced behaviors are typically expressed through scripts that are embedded in web pages. The Mozilla project's XML-based user interface language (XUL) pushes these two technologies even further and relies on them to express an application's entire user interface.

Based on our experiences with *one.world*, which suggest that application-specific policies can be directly implemented on top of our architecture's services, we believe that a similar approach can be applied to the development of pervasive applications. For example, this approach can be used to specify policies for migrating pervasive applications or data integrity constraints for replicated storage. The key insight is that a declarative specification can provide a concise description of a system's properties, which can then automatically be translated into appropriate actions. This approach thus represents a push towards specifying a system's goals instead of programming its behaviors. In effect, it treats a pervasive systems platform, such as *one.world*, as the assembly language for implementing complex behaviors. As such, it holds the potential to significantly simplify the development of complex systems.

Acknowledgments

In addition to the authors of this paper, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Daniel Cheah, Ben Hendrickson, Tom Anderson, Gaetano Borriello,

Steven Gribble, and David Wetherall contributed to the development of *one.world*. More information on our architecture, including a source release, is available at <http://one.cs.washington.edu>.

References

- 38.1 K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.
- 38.2 L. Arnstein, R. Grimm, C.-Y. Hung, J. H. Kang, A. LaMarca, S. B. Sigurdsson, J. Su, and G. Borriello. Systems support for ubiquitous computing: A case study of two implementations of Labscape. In *Proceedings of the 2002 International Conference on Pervasive Computing*, Zurich, Switzerland, Aug. 2002.
- 38.3 R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. Programming for pervasive computing environments. Submitted for publication, Jan. 2002.
- 38.4 M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, Sept. 1991.