System Support for Pervasive Applications

ROBERT GRIMM
New York University
and
JANET DAVIS, ERIC LEMAR, ADAM MACBETH, STEVEN SWANSON,
THOMAS ANDERSON, BRIAN BERSHAD, GAETANO BORRIELLO,
STEVEN GRIBBLE, and DAVID WETHERALL
University of Washington

Pervasive computing provides an attractive vision for the future of computing. Computational power will be available everywhere. Mobile and stationary devices will dynamically connect and coordinate to seamlessly help people in accomplishing their tasks. For this vision to become a reality, developers must build applications that constantly adapt to a highly dynamic computing environment. To make the developers' task feasible, we present a system architecture for pervasive computing, called *one.world*. Our architecture provides an integrated and comprehensive framework for building pervasive applications. It includes services, such as discovery and migration, that help to build applications and directly simplify the task of coping with constant change. We describe our architecture and its programming model and reflect on our own and others' experiences with using it.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; D.4.1 [**Operating Systems**]: Process Management; D.4.2 [**Operating Systems**]: Storage Management; D.4.4 [**Operating Systems**]: Communications Management—*Network communication*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures, patterns*

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Asynchronous events, checkpointing, discovery, logic/operation pattern, migration, *one.world*, pervasive computing, structured I/O, tuples, ubiquitous computing

This work was funded in part under DARPA contracts E30602-98-1-0205 and N66001-99-2-892401. Davis and Swanson were partially supported by NSF fellowships. Davis, Grimm, and Lemar were partially supported by Intel Corporate internships. Grimm was also supported by IBM Corporation and Intel Foundation graduate fellowships.

Authors' addresses: R. Grimm, Department of Computer Science, New York University, 715 Broadway, Room 711, New York, NY 10003; email: rgrimm@cs.nyu.edu; J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195; email: {jlnd,elemar,macbeth,swanson,tom,bershad,gaetano,gribble,djw}@cs.washington.edu.

1. INTRODUCTION

In this article, we explore how to build applications for pervasive computing environments. Pervasive, or ubiquitous, computing [Weiser 1991] has the potential to radically transform the way people interact with computers. It is motivated by the observation that computing and networking technologies are becoming increasingly powerful and affordable, with the result that a wide variety of computing devices can be deployed throughout our living and working spaces. These devices coordinate with each other and network services [Dertouzos 1999], with the goal of providing people with universal access to their information and seamlessly assisting them 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. So, instead of manually managing their computing environment by, for example, copying files between devices or converting between data formats, users "simply" access their applications and their data whenever and wherever they need.

With its vision of ubiquitous information access, pervasive computing significantly impacts how computing devices are deployed and how people interact with the resulting computing infrastructure. First, in addition to well-administered and -controlled computing laboratories and server rooms, computing devices of various sizes and capabilities are now everywhere, often embedded in places not typically associated with computing, such as living rooms or biology laboratories. Second, in contrast to conventional computing environments, people focus on their activities and not on the computers. For example, in a biology laboratory, researchers focus on their experiments and not on the computing devices used to capture experimental results, such as digital pipettes or incubators. Third, tasks often last days and may span many devices, people, and places. Moreover, task requirements may change frequently. For example, biology experiments often take hours, if not days, to complete, and involve many collaborating researchers working at different laboratory stations. As an experiment progresses, researchers may schedule additional steps, for instance, to determine whether an unexpected outcome was caused by contaminants.

The key challenge for developers is to build pervasive applications that continuously adapt to such a highly dynamic environment and continue to function even if people move through the physical world and if the surrounding network infrastructure provides only limited services. However, existing approaches to building distributed applications, including client/server or multitier computing, are ill-suited to meet this challenge. The fundamental problem is that these approaches try to hide distribution and rely on technologies, such as remote procedure call (RPC) packages [Birrell et al. 1982] or distributed file systems [Levy and Silberschatz 1990], that extend single-node programming methodologies to distributed systems. Because these technologies hide remote interactions, favor static composition through programmatic interfaces, and often encapsulate data and functionality in the form of objects, they make it hard to anticipate failures, to extend applications, and to share and search data. Consistent with the push towards hiding distribution, applications built on top of these technologies

tend to be structured like single-node applications and assume an execution environment where resources are constant and continuously available.

As a result, users are forced to "stitch up the seams" and need to explicitly reconfigure their computers every time the execution environment changes. For example, with today's wireless networking technologies, people need to manually adapt their computers every time they enter a different network. Existing systems and applications have no notion of "entering a new network" and thus need to be explicitly configured with the wireless network name and access key, to say nothing of necessary file servers or close-by printers. However, forcing users to adapt is impractical and, fundamentally, antithetical to the vision of pervasive computing.

To mitigate this situation, we present a system architecture, called *one.world*, that provides an integrated framework for building adaptable applications. Our work is motivated by the insight that, in direct opposition to conventional distributed systems, system support for pervasive applications must expose distribution rather than hide it. That way, applications can see change and then adapt to it instead of forcing users to repeatedly reconfigure their systems. More specifically, system support for pervasive applications must meet three 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 change 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 *recognize sharing as the default*.

Our architecture, *one.world*, represents a first stab at exploring how to address these three requirements. *one.world* is layered over a traditional operating system, such as Windows or Linux. It exposes a simple programming model that relies on tuples (possibly nested records of name/value pairs) for all data, thus making it easy to share data, and asynchronous events for all communications, thus providing a well-defined mechanism for notifying applications of change. Like any distributed system, it has facilities for managing processes, storage, and point-to-point communications. More importantly, it provides a set of services, such as discovery and migration, that directly simplify the task of coping with constant change. Our architecture reuses existing operating system technologies where appropriate, and innovates where necessary; the focus is to provide an integrated and comprehensive framework for building pervasive applications.

We have validated our architecture by supporting the Labscape project [Arnstein et al. 2002] in porting their digital biology laboratory assistant to *one.world* and by building our own utilities and applications on top of *one.world*—notably, Emcee, our user and application manager, and Chat, a text and audio messaging system. Based on these experiences, we show that our architecture (1) enables others to successfully build pervasive applications, (2) is not

significantly harder to program than with conventional programming styles, (3) is sufficiently complete to support additional services and utilities on top of it, and (4) has acceptable performance, with applications reacting quickly to changes in their runtime context. Our evaluation thus validates our approach. At the same time, our own and others' experiences with using *one.world* suggest several opportunities for future research, including the need for data models that are general and supported by a wide range of platforms and the need for user interfaces that scale across a wide range of devices. Additionally, while our architecture has been designed to support reference monitors and auditing, security for pervasive computing environments—specifically the authentication of users, devices, and applications and the expression of appropriate security policies—remains an important, open issue.

The contributions of this article are threefold. First, we discuss the shortcomings of existing distributed systems technologies, which typically extend single-node programming methodologies, and present an alternative approach that cleanly exposes distribution, thus providing a more appropriate foundation for building pervasive applications. Second, we present a system architecture, *one.world*, that embodies this approach, and we provide a detailed description of its programming model. Third, we present the results of a thorough experimental evaluation of *one.world*, which is based on our own as well as others' experiences with our architecture, relay lessons learned, and identify opportunities for future research.

This article is structured as follows. In Section 2, we motivate our work and introduce our approach to building pervasive applications. Section 3 provides an overview of our architecture. Next, Section 4 describes *one.world*'s programming model in detail. We present the results of our experimental evaluation in Section 5 and reflect on our own and others' experiences with using *one.world* in Section 6. Section 7 reviews related work. Finally, Section 8 concludes this article.

2. MOTIVATION AND APPROACH

From a systems viewpoint, the pervasive computing space presents the considerable challenge of a large and highly dynamic distributed computing environment. However, existing approaches to building distributed systems do not provide adequate support for addressing this challenge and fall short along three main axes.

First, many existing distributed systems seek to hide distribution and, by building on distributed file systems [Levy and Silberschatz 1990], or remote procedure call (RPC) packages [Birrell et al. 1982], mask remote resources as local resources. This transparency simplifies application development, since accessing a remote resource is just like performing a local operation. However, it also comes at a cost in service quality and failure resilience. By presenting the same interface to local and remote resources, transparency encourages a programming style that ignores the differences between the two, such as network bandwidth [Muthitacharoen et al. 2001], and treats the unavailability of a resource or a failure as an extreme case. But in an environment where people and

devices keep on coming and going, change is inherent and the unavailability of some resource is frequent.

Second, RPC packages and distributed object systems, such as Legion [Lewis and Grimshaw 1996] or Globe [van Steen et al. 1999], compose distributed applications through programmatic interfaces. Just like transparent access to remote resources, composition at the interface level simplifies application development. However, composition through programmatic interfaces also leads to a tight coupling between major application components because they directly reference and invoke each other. As a result, it is unnecessarily hard to add new behaviors to an application. Extending a component requires interposing on the interfaces it uses, which requires extensive operating system support [Jones 1993; Pardyak and Bershad 1996; Tamches and Miller 1999] and is unwieldy for large or complex interfaces. Furthermore, extensions are limited by the degree to which extensibility has been designed into the application's interfaces.

Third, distributed object systems encapsulate both data and functionality within a single abstraction, namely objects. Yet again, encapsulation of data and functionality extends a convenient paradigm for single-node applications to distributed systems. However, by encapsulating data behind an object's interface, objects complicate the sharing, searching, and filtering of data. In contrast, relational databases define a common data model that is separate from behaviors and thus make it easy to use the same data for different and new applications. Furthermore, objects as an encapsulation mechanism are based on the assumption that data layout changes more frequently than an object's interface, an assumption that may be less valid for a global distributed computing environment. Increasingly, many different applications manipulate the same data formats, such as XML [Bray et al. 1998]. These data formats are specified by industry groups and standard bodies, such as the World Wide Web Consortium, and evolve at a relatively slow pace. In contrast, application vendors compete on functionality, leading to considerable differences in application interfaces and implementations and a much faster pace of innovation.

Not all distributed systems are based on extensions of single-node programming methodologies. Notably, the World Wide Web does not rely on programmatic interfaces and does not encapsulate data and functionality. It is built on only two basic operations, GET and POST, and the exchange of passive, semi-structured data. In part due to the simplicity of its operations and data model, the World Wide Web has successfully scaled across the globe. Furthermore, the narrowness of its operations and the uniformity of its data model have made it practical to support the World Wide Web across a huge variety of devices and to add new services, such as caching [Chankhunthod et al. 1996; Tewari et al. 1999], content transformation [Fox et al. 1997], and content distribution [Johnson et al. 2000].

However, from a pervasive computing perspective the World Wide Web also suffers from three significant limitations. First, just like conventional distributed systems, it places the burden of adapting to change on users, for example, by making them reload a page when a server is unavailable because it is overloaded or inaccessible. Second, it requires connected operation for any

use other than reading static pages. Finally, it does not seem to accommodate emerging technologies that are clearly useful for building adaptable applications, such as service discovery [Adjie-Winoto et al. 1999; Arnold et al. 1999; Czerwinski et al. 1999] and mobile code [Thorn 1997]. While Java applets are a form of mobile code, they are only active while the corresponding page is displayed and, by default, can only communicate with the originating server. As a result, they are basically limited to enlivening web pages and implementing site-specific chat clients.

2.1 The Unique Requirements of Pervasive Computing

The inadequacy of existing distributed systems raises the question of how to structure system support for pervasive applications. On one side, extending single-node programming models to distributed systems leads to the shortcomings discussed above. On the other side, the World Wide Web avoids several of the shortcomings but is too limited for pervasive computing. To help define a better alternative, we identify the three unique requirements of pervasive computing.

REQUIREMENT 1. *Embrace contextual change.*

As people move through the physical world, the execution context of their applications changes all the time. It is impractical to ask users to manually manage these changes, such as entering a new wireless network name and access key every time they enter a different network. Systems thus need to expose contextual changes rather than hiding distribution, so that applications can implement their own strategies for handling changes and spare the users from doing so. Event-based notification or callbacks are examples of suitable mechanisms. At the same time, systems need to provide primitives that simplify the task of adequately reacting to change. Examples for such primitives include "checkpoint" and "restore" to simplify failure recovery, "move to a remote node" to follow a user as she moves through the physical world, and "find matching resource" to discover suitable resources on the network, such as nearby instruments in a biology laboratory or other users with whom to exchange messages.

REQUIREMENT 2. *Encourage ad hoc composition.*

As people use different devices in different locations, they expect that applications and devices just plug together. It is impractical to ask users to manually perform the composition. Consequently, systems should make it easy to compose applications, services, and devices at runtime. In particular, installing a user's applications on a device must be easy. Furthermore, interposing on an application's interactions with other applications and network services must be simple. Interposition makes it possible to dynamically change the behavior of an application or add new behaviors without changing the application itself. This is particularly useful for complex and reusable behaviors, such as replicating an application's data or deciding when to migrate an application.

REQUIREMENT 3. *Recognize sharing as the default.*

In essence, pervasive computing strives to make information accessible anywhere and anytime. It is impractical to ask users to manage the corresponding files (by, for example, moving them between different devices) and to convert between different data formats. Systems thus need to make it easy to access saved information and to share information between different applications and devices. Ease of sharing is especially important for services that search and filter large amounts of data. At the same time, data and functionality depend on each other, for example, when migrating an application and its data. Systems thus need to include the ability to group data and functionality while still making them accessible independently.

Individually, the three requirements have been recognized before. For instance, Bayou [Petersen et al. 1997; Terry et al. 1995] exposes different data values on different devices, and Odyssey [Noble et al. 1997] relies on asynchronous notifications to expose contextual change to applications. Furthermore, as already discussed, the World Wide Web is built on only two basic operations, GET and POST, which greatly simplifies dynamic composition. Finally, the recent move towards expressing all data on the Internet as XML attempts to facilitate sharing. Our approach differs from these efforts in that we advocate addressing all three requirements *at the same time*.

Common to the requirements is the realization, similar to that behind extensible operating systems [Bershad et al. 1995; Engler et al. 1995; Kaashoek et al. 1997], that systems cannot automatically decide how to react to change, because there are too many alternatives. Where needed, the applications themselves should be able to determine and implement their own policies [Saltzer et al. 1984]. As a result, we are advocating a structure different from previous distributed systems, which exposes distribution so that applications can adapt to change instead of users.

At the same time, the three requirements do not preclude the use of established programming methodologies. Embracing contextual change does not prevent us from providing reasonable default behaviors. But it does emphasize that applications must be notified of change. Similarly, encouraging ad hoc composition does not preclude the use of strongly typed APIs. However, it does emphasize the need for simplifying interposition. Finally, recognizing sharing as the default does not preclude the use of object-oriented programming. The ability to abstract data or functionality is clearly useful for structuring and implementing applications. At the same time, ease of sharing, with its emphasis on searching, filtering, and translating data, does suggest that application data and functionality build on distinct abstractions.

Given a system that meets these requirements, application developers can focus on making applications adaptable, and the users, in turn, can focus on their tasks instead of manually adapting their devices and applications. While programming adaptable applications requires developer discipline when compared to conventional distributed systems, it also provides an important opportunity to transform the way people interact with their computers and applications. This approach to building pervasive applications is illustrated in Figure 1.

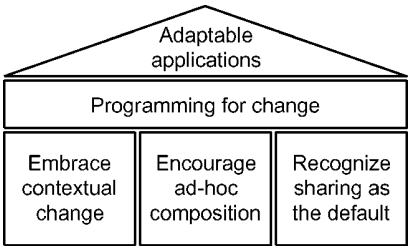

Fig. 1. Illustration of our approach. The three requirements guide the design of our system architecture and make it feasible for application developers to program for change, resulting in adaptable applications.

2.2 Adaptability and Transparency

By exposing distribution, our approach differs from a large class of efforts that have explored how to build services that adapt (largely) transparently to an ever changing execution context. For instance, the Coda file system [Kistler and Satyanarayanan 1992; Mummert et al. 1995] aggressively caches files on clients—hoarding files before they might be accessed—to support disconnected or weakly connected operation. Similarly, the Rover system [Joseph et al. 1995] caches service objects on clients and provides queued RPC to support mobile devices that may only be intermittently connected. The xFS file system [Anderson et al. 1996] automatically distributes file storage, caching, and control across a set of cooperating workstations and thus eliminates the need for dedicated file servers. More ambitiously, the OceanStore project [Kubiatowicz et al. 2000; Rhea et al. 2003] is trying to create a global storage utility, which runs on an untrusted computing infrastructure and automatically moves and replicates data between devices to optimize for locality and availability. Furthermore, the Mobile IP architecture [Ioannidis and Maguire 1993] supports device mobility by automatically forwarding TCP/IP traffic, even if a device is not connected to its home network. Finally, the Barwan networking architecture [Brewer et al. 1998] includes support for transparently switching between wired and wireless networks and for correspondingly switching between transport protocols to ensure that devices remain continuously and reliably connected, independent of their current location.

Common to these efforts is that they focus less on delivering programming methodologies (like the RPC and distributed object systems discussed above) and more on providing particular *services*. As a result, they can focus on making the provided services adaptable. More fundamentally, to be transparent, these efforts also share a drive to contain changes to existing applications and networking infrastructure as much as possible. One important technique employed by many of these efforts is the use of proxies. For example, xFS includes NFS proxies that provide access to its file system data to unmodified clients. Furthermore, Rover includes a web proxy to provide web access to mobile devices while also leaving existing browsers and servers unchanged. Finally, both Mobile IP and Barwan use proxies to isolate protocol additions to the mobile clients and their routers, with Barwan also generalizing proxies under their TACC (for transformation, aggregation, caching, and customization) model [Fox

et al. 1997]. As a result, these services can adapt to a changing execution context, while also being able to transparently interact with legacy systems and applications.

Overall, we believe that these efforts are complimentary to our own approach for two reasons. First, by providing continuous access to important services, such as storage and networking, these efforts certainly lessen the burden of making applications adaptable and thus simplify the development of pervasive applications. In fact, *one.world* also reflects the desire to isolate applications from at least some changes. Notably, the implementation of our architecture's discovery service relies on an automatically elected server and thus transparently adapts to a changing device and network topology. Discovery server elections ensure that the directory of discoverable resources is almost always available while also hiding the directory's location. This implementation trade-off is reasonable, as directory availability is considerably more important to pervasive applications than directory location.

Second, since a system architecture that meets the three requirements has been specifically designed for implementing adaptable programs, we believe that such an architecture can also simplify the implementation of transparently adapting services—an important concern when considering the complexity of services such as xFS or OceanStore. Furthermore, several of the above systems are only transparent *to a degree* and need to expose some changes to applications. For example, the resolution of file conflicts in Coda is at least type-specific if not application-specific [Kumar and Satyanarayanan 1995]. Furthermore, Barwan needs to notify applications that the current network has changed so that they can adapt, for example, the fidelity of streaming audio or video to match available bandwidth. Clearly, a system architecture that follows our approach provides a convenient framework for exposing such changes.

2.3 The Biology Laboratory as an Example Application Domain

To illustrate the three requirements central to our approach to building pervasive applications, we now introduce the digital biology laboratory. Unlike the scenarios presented in Esler et al. [1999] and Weiser [1991], the digital laboratory does not illustrate the full potential of pervasive computing. However, it addresses a real need of real people—performing reproducible biology experiments. Furthermore, as discussed in Section 5.5, the digital laboratory has been implemented on top of *one.world* by the University of Washington's Labscape project and has been deployed at the Cell Systems Initiative [Arnstein et al. 2002]. As a result, it provides an apt example for a pervasive application and the three requirements.

As already mentioned, the goal of researchers working in a biology laboratory is to perform reproducible experiments. 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'

Fig. 2. A workbench in a biology laboratory. Notice how the touchscreen on the right hand side becomes just another instrument on this workbench rather than being the focus of attention.

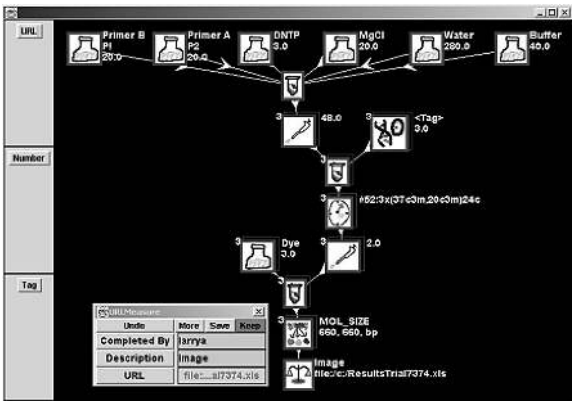

Fig. 3. A screenshot of Labscape's user interface. Each experiment is represented as an experimental flowgraph, or *guide*. The individual icons represent different experimental steps, and the arrows represent ordering constraints.

movements, and touchscreens to display experimental data close to the researchers (see Figure 2 for a workbench in the digital laboratory). As a result, biologists in the digital laboratory have more complete records of their experiments and can more easily share results with their colleagues.

A fundamental feature of the digital laboratory is that experimental data follows a researcher as she moves through the laboratory. Furthermore, the data can follow her as she leaves the laboratory, for example, so that she can review a day's results on her tablet computer while taking the commuter train home. At the same time, there is no need to move the entire digital laboratory application as the researcher moves through the physical world. Rather, only a small component for capturing and displaying experimental data needs to follow the researcher. Eventually, all data is forwarded to a centralized repository, making it possible, for example, to mine the data of several experiments.

Figure 3 shows a snapshot of the capture and display component's user interface, which is also called a *guide* and has been developed by the Labscape

project through user interface studies with actual biology researchers. Each experiment is represented as a flowgraph. The individual icons represent different experimental steps and the arrows represent ordering constraints between the steps. Originally, the guide functions as a plan for the experiment to be performed. A researcher can either select a flowgraph from a library of existing flowgraphs or create her own (which may be based on an existing flowgraph). As the researcher performs a step, she annotates the corresponding icon with the results of that step, or, if she is using digital instruments, they are automatically annotated. Over time, the flowgraph thus becomes a record of the experiment.

The three requirements of pervasive computing—change, ad hoc composition, and pervasive sharing—show up in the digital laboratory as follows:

Embrace contextual change. Biology laboratories are organized into task-specific work areas, often centered around a specific instrument, such as a centrifuge or incubator, and biologists often move between work areas while working on the same experiment. As a result, a researcher's location changes need to be exposed to her guide, so that it can automatically follow her to the corresponding touchscreens.

Encourage ad hoc composition. As the researcher moves through the biology laboratory, her guide needs to transparently connect to the digital instruments in her current work area and incorporate readings into the experimental flowgraph. Furthermore, as outside researchers visit the laboratory, their PDAs or laptops need to be automatically integrated into the digital laboratory so that the researchers can exchange and review experimental results.

Recognize sharing as the default. Biology experiments often last hours, if not days, and biologists multitask between several experiments or focus on performing a single step for many experiments in a row. As a result, it must be easy to switch between different experiments, to hand off experiments between researchers, and to compare different experiments.

As a pervasive application, the digital laboratory application needs to be considerably more seamless and adaptable than conventional distributed applications. At the same time, it also illustrates an important property of pervasive computing environments: their human scale. In particular, the digital laboratory only needs to scale to a limited number of concurrent users, as only so many people can work in the same laboratory at the same time. Furthermore, the digital laboratory application typically needs to adapt at a human time scale. A researcher walking from one work area to another thus leaves a relatively large timespan (compared to the microsecond latencies often considered in distributed systems work) for migrating the researcher's guide and connecting to close-by instruments.

2.3.1 *Short-comings of the Status Quo.* To emphasize that pervasive applications are not just conventional distributed applications, we now consider the limitations of conventional systems when implementing the digital laboratory. First, it is hard to move between devices. Even with existing networked

application support, such as the X Window System [Nye 1995] or roaming profiles in Windows [Tulloch 2001], users have to manually log into a machine, start their applications, and load the necessary data. Second, it is hard to connect to different instruments as researchers move between work areas. Conventional systems focus on providing point-to-point communications (with TCP being the most prominent example) and lack facilities for dynamically discovering and connecting to close-by instruments without explicit, manual configuration. Finally, it is hard to share data. On one hand, file systems support only coarse-grained sharing—remember that biology experiments consist of many steps, with each step only generating a small amount of data. On the other hand, databases are difficult to set up and administer, typically requiring dedicated facilities and staff.

Based on similar observations, several efforts have explored how to layer additional middleware for building adaptable applications onto existing distributed systems. Sun's Jini is probably the most popular example for such a middleware platform [Arnold et al. 1999]. Like *one.world*, it supports distributed events, tuple storage, and service discovery. Unlike *one.world*, it is layered on top of Java RMI [Sun Microsystems 2002], a traditional distributed object system, and thus inherits RMI's limitations. In particular, Jini requires a statically configured infrastructure to run its discovery server. Furthermore, it requires an overall well-behaved computing environment because it relies on transparent and synchronous remote invocations, does not provide isolation between applications running within the same Java virtual machine, and links objects on different devices with each other through distributed garbage collection. In other words, Jini is inherently limited because it builds on a conventional distributed system. To be effective, system support for pervasive applications must be designed from the ground up to meet the three requirements: change, ad hoc composition, and pervasive sharing.

3. ARCHITECTURE

With the three requirements in place, we now introduce *one.world* and its services. Our architecture is structured according to the following four principles. First, bias a set of foundation services to directly address the three requirements: change, ad hoc composition, and pervasive sharing. Second, express specific system services in terms of the foundation services and make them available as common application building blocks. Third, employ a classic user/kernel split, with foundation and system services provided by the kernel, and libraries, system utilities, and applications running in user space. Finally, remain neutral on other issues, such as whether to implement applications as client/server or peer-to-peer applications. The resulting organization is illustrated in Figure 4. We now present the individual services as well as the provided library support in more detail.

3.1 Foundation Services

The four foundation services directly address the three requirements of change, ad hoc composition, and pervasive sharing. First, a *virtual machine*, such as

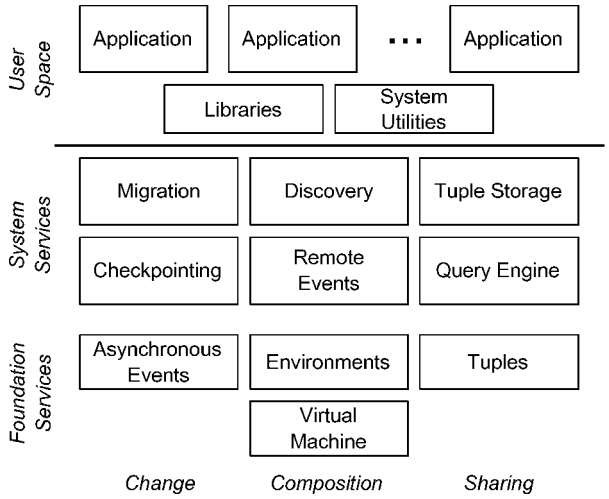

Fig. 4. Overview of *one.world*'s architecture. Foundation and system services are part of the kernel, while libraries, system utilities, and applications run in user space.

the Java virtual machine [Lindholm and Yellin 1999] or Microsoft's Common Language Runtime [Thai and Lam 2002], provides a common execution environment across all devices and hardware platforms.[1] Since developers cannot possibly predict all the devices their applications will run on, the virtual machine ensures that applications and devices are composable. Second, *tuples* define a common data model, including a type system, for all applications and thus make it easy to share data. They are records with named fields and are self-describing in that an application can dynamically inspect a tuple's fields and their types. Third, all communications in *one.world*, whether local or remote, are through *asynchronous events*; applications are composed from components that exchange events through imported and exported event handlers. Events make change explicit to applications, with the goal that applications adapt to change instead of forcing users to manually reconfigure their devices and applications.

Finally, *environments* are the central mechanism for structuring and composing applications. They serve as containers for stored tuples, application components, and other environments, and form a hierarchy with a single root per device. Each application consists of at least one environment, in which it runs and stores its persistent data. However, applications are not limited to a single environment and may span several, nested environments. Comparable to processes in conventional operating systems, environments provide protection and isolate applications from each other and from *one.world*'s kernel, which is hosted by each device's root environment. Environments also are an important mechanism for dynamic composition: an environment controls all nested environments and can interpose on their interactions with the kernel and the outside world. Environments thus represent a combination of the roles served

[1] *one.world* is implemented in Java. At the same time, its implementation does not rely on any features that are unique to Java, and it could be implemented on a different virtual machine platform, such as Microsoft's Common Language Runtime.

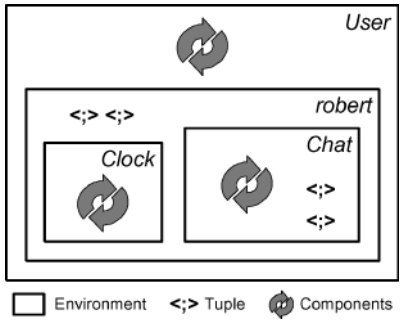

Fig. 5. Illustration of an example environment hierarchy. The *User* environment hosts the Emcee application and has one child, named *robert*, which stores several tuples representing that user's preferences. The *robert* environment in turn has two children, named *Clock* and *Chat*. The *Clock* environment only contains active application components, while the *Chat* environment, in addition to hosting the Chat application, also stores tuples representing the music being broadcast by Chat.

Table I. Application Needs and Corresponding System Services

Applications need to . . .	***one.world* provides . . .**
Search	Query engine
Store data	Structured I/O
Communicate	Remote events
Locate	Discovery
Fault-protect	Checkpointing
Move	Migration

by file system directories and nested processes [Brinch Hansen 1970; Ford et al. 1996; Tullmann and Lepreau 1998] in other operating systems. Figure 5 shows an example environment hierarchy.

3.2 System Services

In addition to the foundation services, *one.world* provides a set of system services that serve as common application building blocks. Table I summarizes common application needs and the corresponding system services.

The *query engine* provides the ability to search tuples by instantiating filters. Queries support the comparison of a constant to the value of a field, the comparison to the type of a tuple or field, and negation, disjunction, and conjunction. *Structured I/O* lets applications access stored tuples in environments. It supports the writing, reading, querying, and deleting of tuples. The structured I/O operations are atomic so that their effects are predictable and can optionally use transactions to group several operations into one atomic unit. The query engine and structured I/O simplify data access because applications can directly access relevant data items.

Remote event passing (REP) forwards events to remote services and is *one.world*'s basic mechanism for communicating across the network. Consistent with our push towards exposing distribution and in contrast to RPC or distributed object systems, remote communications in *one.world* are explicit. To use REP, services export event handlers under symbolic descriptors, that is,

tuples, and clients send events by specifying the symbolic receiver. *Discovery* locates services with unknown locations. It supports a rich set of options, including early and late binding [Adjie-Winoto et al. 1999] as well as anycast and multicast, and is fully integrated with REP, resulting in a simple, yet powerful API. Discovery is especially useful for applications that migrate or run on mobile devices and need to find local resources, such as a close-by digital instrument.

Checkpointing captures the execution state of an environment tree and saves it as a tuple, making it possible to later revert the environment tree's execution state. Checkpointing simplifies the task of gracefully resuming an application after it has been dormant or after a failure, such as a device's batteries running out. *Migration* provides the ability to move or copy an environment and its contents, including stored tuples, application components, and nested environments, either locally or to another device. It is especially useful for applications that follow a person from shared device to shared device as she moves through the physical world.

3.3 Library Support

Outside of *one.world*'s kernel, our architecture provides additional, user-level library support for implementing pervasive applications. The libraries include functionality for constructing an application's user interface and for the timed execution of event handlers. More importantly, *operations* help manage asynchronous interactions. They are based on what we call the logic/operation pattern. This pattern structures applications into logic—computations that do not fail, such as creating and filling in a message, and operations—interactions that may fail, such as sending the message to its intended recipients. Operations simplify such interactions by keeping the state associated with event exchanges and by providing automatic timeouts and retries.

3.4 The Big Picture

Pulling back, Figure 6 illustrates the big picture behind our architecture. The basic idea is that all the different devices in a pervasive computing environment run the same system platform, namely *one.world*. While individual devices may provide additional services over those supplied by our architecture, applications can rely on the same basic operating environment on every device (modulo differences in CPU speed and memory/storage capacity). Each device is independent of other devices and need not be connected with (all) other devices. Furthermore, each device may be administered separately.

The different applications running on the different devices interact with each other by exchanging tuples and events. When communicating with other applications, an application may not necessarily know how many applications it is communicating with and where those applications are located (hence the heap of data in the figure). Furthermore, as people move through the physical world, applications follow by migrating from device to device to device. Though, as illustrated by the digital biology laboratory, we expect that many applications only need to migrate a relatively small component. Consistent with our principle of remaining neutral on how to structure applications, the migrating

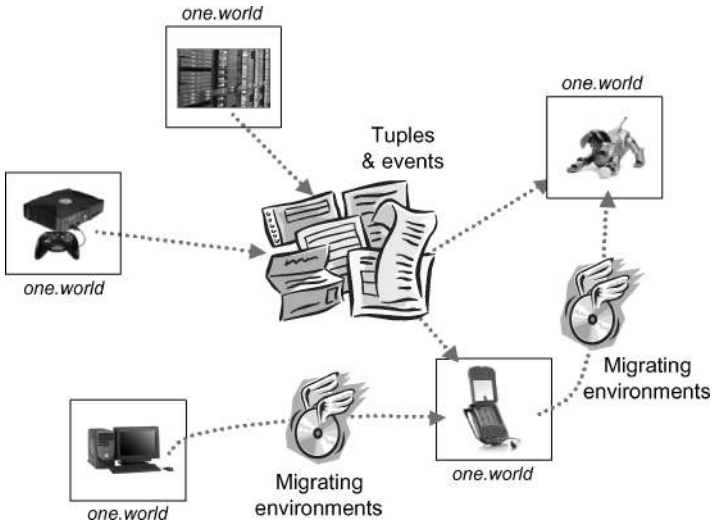

Fig. 6. The big picture. All the different devices in a pervasive computing environment run the same system platform, *one.world*. The different applications exchange tuples and events between each other and migrate from one device to another.

component may rely on additional server-based or peer-to-peer services for its full functionality.

For example, in the case of the digital biology laboratory, the digital instruments send events describing performed operations to a researcher's guide, and the guide, in turn, forwards experimental results to the centralized data repository. As the researcher is switching between work areas, her guide follows her by migrating from touchscreen to touchscreen to touchscreen while the data repository remains in place. Similarly, when using Emcee and Chat, a user's Chat application sends text and audio messages by sending events to other people's instances of Chat. As the user moves between rooms, Emcee migrates her Chat application so that Chat is always running on a device close to that person.

3.5 *one.world* and Distributed Systems Concerns

Like any distributed system, *one.world* must address several distributed systems issues, such as how to provide processes, storage, and communications. Table II lists the most important issues and relates them to the corresponding features in our architecture. The table also lists the specific sections that discuss these features in detail, thus serving as an index into the programming model section of this article. The *request/monitor mechanism* listed in the table is the interposition mechanism enabled by nested environments; it lets an outer environment interpose on all interactions of nested environments with the kernel and the outside world.

4. PROGRAMMING MODEL

We now explore *one.world*'s programming model in detail. We describe our architecture's services and their operations, give code examples, and explain our

Table II. *one.world* and Distributed Systems Concerns

Issue	Feature	Section
Namespaces	Environments contain applications and their persistent data.	4.1
	Protection domains limit access to direct references.	4.1
	The environment hierarchy limits access to nested environments.	4.1
Data management	Tuples represent all data.	4.2
	The query engine searches tuples.	4.2.1
	Structured I/O persistently stores tuples in environments.	4.2.2
Execution model	Applications are composed from components that exchange asynchronous events.	4.3
	The request/monitor mechanism provides the system call interface.	4.3.1
	Operations manage event exchanges, notably those with event handlers outside an application.	4.3.2
Communication model	Discovery locates remote receivers by their descriptions.	4.4
	Remote event passing sends events to remote receivers.	4.4
Application persistence	Checkpointing captures an application's execution state, and migration moves or copies an application.	4.5
Security	Protection domains isolate applications.	4.1
	The request/monitor mechanism can be used to implement reference monitors and auditing.	4.3.1
Resource allocation	The request/monitor mechanism can be used to interpose on requests for system services.	4.3.1
Extensibility	The request/monitor mechanism can be used to add new services.	4.3.1

Issue specifies the distributed systems concern. *Feature* describes the corresponding *one.world* service. *Section* lists the programming model section discussing that feature.

design decisions. In contrast to the previous section, which is organized around horizontal slices through our architecture—foundation services, system services, and library support, this section is organized around conventional systems concerns, focusing on the perspective of application developers becoming familiar with our architecture. In particular, we start with namespaces in Section 4.1 and explore environments as containers for applications, their persistent data, and other environments. We follow with data management in Section 4.2 and discuss tuples, the query language and engine, as well as structured I/O. In Section 4.3, we explore *one.world*'s execution model and present events, the request/monitor mechanism, and operations. Next, in Section 4.4, we discuss the communication model and describe discovery and remote event passing. Finally, in Section 4.5, we explore application persistence and present checkpointing and migration. We do not further discuss virtual machines, as our architecture directly builds on existing virtual machine technology.

In summary, an application in *one.world* consists of an environment, which acts as the namespace for the application's objects and can include code, data, and other environments. The application's code executes in response to asynchronous events, which may be generated by the system, the application itself, or by other applications, potentially on other devices. Events are delivered through a rich event delivery interface. The application's data is stored in an

associative tuple store, which is part of the application's environment. The application's execution state can be checkpointed and stored in its environment so that the application can later be reverted to the saved state. The application can also be migrated to a different device, which either moves the application's environment and all its contents to that device or creates a copy on the remote device. *one.world* itself executes as a set of kernel services, available to, and mediating, all applications running on a device.

4.1 Namespaces

Comparable to processes in conventional operating systems, environments host applications and isolate them from each other through protection domains. By default, each environment represents its own protection domain; though, a protection domain may span several, nested environments. To enforce isolation, all data is copied between protection domains, while still allowing for the exchange of event handlers so that applications can communicate with each other. Operations on environments and access to an environment's tuple storage are limited to the requesting environment and its descendants, thus making it possible to limit an application's effects to its subtree, which is important as potentially untrusted applications move from one device to another.

Environments provide structure not only by isolating applications from each other, but also by grouping application functionality and persistent data within the same container. The grouping of functionality and data enables *one.world* to load an application's code from its environment and to store the application's checkpoints with the application. More importantly, it simplifies the development of pervasive applications that follow a user through the physical world, as an application and its data, including code and checkpoints, can be migrated in a single operation. At the same time, environments always maintain a clear separation between functionality and data, which can be accessed independently and, unlike objects, are not hidden behind a unifying interface.

In addition to providing structure, environments provide control through nesting: an outer environment has full control over an inner environment, including the ability to interpose on the inner environment's interactions with the kernel and the outside world. Nesting thus makes it possible to easily factor important pervasive computing features, such as the logic to control migration and the ability to synchronize data with other devices, out of an application and reusing that functionality across several applications. To exploit nesting for this purpose, the reusable functionality is provided by an outer environment, and the application relying on that functionality is placed into an inner environment. Note that, as described in Ford et al. [1996] and Tullmann and Lepreau [1998], nesting also facilitates hierarchical resource controls on CPU time and main memory. While clearly important, we have not further explored the use of environment nesting for managing these resources in *one.world*.

Table III summarizes the environment operations. The majority of these operations work as expected and are used to create and delete environments and to start and stop applications. The *checkpoint*, *restore*, *move*, and *copy* operations

Table III. The Environment Operations

Operation	Explanation
create	Create a new environment.
rename	Rename an environment.
load	Load an application into an environment.
activate	Activate the application.
terminate	Terminate the application.
unload	Unload the application.
destroy	Delete the environment and all its contents.
move	Move the environment and all its contents.
copy	Copy the environment and all its contents.
checkpoint	Checkpoint the environment.
restore	Restore a previously captured checkpoint.

Operation specifies the environment operation, and *explanation* describes the operation.

are used for checkpointing and migration and are described in Section 4.5. To perform an operation, an application specifies the operation, the targeted environment, and any additional arguments as necessary. Environments are named by either a globally unique identifier [Leach and Salz 1998] (GUID) or a human-readable path name, which, like a path name in Unix, is composed of individual environments' names separated by slashes ('/'). An environment's GUID cannot be changed after creation, so that it can be used as a unique reference for that environment. In contrast, an environment's human-readable name, just like a directory name in conventional file systems, can be changed after creation through the *rename* operation to accommodate changing user needs. *one.world*'s kernel runs in a device's root environment, which, just like the root directory in Unix, is named "/".

4.2 Data Management

Data management in *one.world*, that is, the ability to query, store, and exchange information, is based on tuples. Tuples define the common data model, including the type system, for applications running in our architecture. They are self-describing, mutable records with named and (usually) typed fields. Valid field types include numbers, strings, and arrays of basic types, as well as tuples, thus allowing tuples to be nested within each other. Arbitrary objects can be stored in a tuple in the form of a special container that encapsulates a serialized representation of the object.

By providing a common, structured data model, tuples enable our architecture's data management services, notably the query engine and structured I/O. As a result, tuples let pervasive applications directly encode and exchange the information they manage. They also obviate the need for separate internal and external representations and for translating between different data formats, generally simplifying the sharing of information. Consider, for example, a personal information management application. It can directly encode a user's appointments, contacts, notes, and messages as tuples and, through the query engine and structured I/O described below, search, store, and exchange that data. As a result, it becomes easier to make a user's data available throughout

```
public abstract class Tuple {
  // The ID and metadata fields.
  public Guid         id;
  public DynamicTuple metaData;

  // Programmatic access to a tuple's fields.
  public final Object get(String name) {...}
  public final void   set(String name, Object value) {...}
  public final Object remove(String name) {...}
  public final List   fields() {...}
  public final Class  getType(String name) {...}

  // Validation of a tuple's semantic constraints.
  public void validate() throws TupleException {...}

  // A tuple's human-readable representation.
  public String toString() {...}
}
```

Fig. 7. Definition of a tuple. Each tuple has an ID field to support symbolic references and a metadata field to support application-specific annotations. A set of methods provides programmatic access to a tuple's fields, and additional methods support the expression of semantic constraints and a human-readable representation.

her living and working spaces and to synchronize between different devices, applications, and people.

To capture the structure of application data, tuples are statically declared and strongly typed. A tuple has a fixed set of fields with specific types, and the overall tuple has a type. However, our architecture also includes a special tuple, called `DynamicTuple`. In contrast to other tuples, the fields of a dynamic tuple can be dynamically added and removed and are dynamically typed, that is, they can have any allowable field type. As a result, dynamic tuples are more flexible, but do not offer typing guarantees. They are useful for representing ad hoc data, such as another tuple's metadata, or for prototyping data records during application development.

All tuples share the same base class shown in Figure 7. Each tuple has an ID field specifying a GUID to support symbolic references, as well as a metadata field to support application-specific annotations. Each tuple also has a set of methods to programmatically reflect its structure and to access its data, thus allowing applications to inspect and access data items with unknown types. The accessor methods are final and are implemented using reflection (with the `remove()` method only working for dynamic tuples). Finally, each tuple has a `validate()` method to validate its semantic constraints (i.e., to determine whether a tuple's field values are consistent with each other) and a `toString()` method to produce a human-readable representation. In contrast to the accessor methods, individual tuple classes can override the latter methods to express their own semantic constraints and human-readable representation.

Table IV. The Structured I/O Operations

Operation	Argument	Explanation
put	Tuple	Write the specified tuple.
read	Query	Read a single tuple matching the specified query.
query	Query	Read all tuples matching the specified query.
listen	Query	Observe all tuples that match the specified query as they are written.
delete	ID	Delete the tuple with the specified ID.

Operation specifies the structured I/O operation. *Argument* specifies how tuples are selected for that operation. *Explanation* describes the operation.

4.2.1 *Query Language and Engine.* So that applications can easily search and filter data, such as a user's appointments or contacts, our data model also defines a common query language for tuples. That language is designed to support complex queries over both the structure (i.e., types) and the contents (i.e., values) of tuples and is used by the APIs for the structured I/O and discovery services. Queries support the comparison of a constant to the value of a field, including the fields of nested tuples, the comparison of a type to the declared or actual type of a tuple or (nested) field, and negation, disjunction, and conjunction. Since queries are data themselves, they are also expressed as tuples.

An example query in our architecture's query language is shown in Figure 14. It consists of several, nested `Query` tuples, which express a type comparison (as indicated by the `Query.COMPARE_HAS_SUBTYPE` constant), a value comparison (as indicated by the `Query.COMPARE_EQUAL` constant), and a conjunction (as indicated by the `Query.BINARY_AND` constant). The overall query matches tuples of type `UserDescriptor` whose `user` field equals the value of the `fetchUser` variable.

The query engine processes queries over tuples, as expressed in our architecture's query language. To use the query engine, services and applications instantiate a filter for a specific query (such as the one referenced above) and then feed tuples to the filter. Tuples matching the query are passed through and tuples not matching the query are dropped.

4.2.2 *Structured I/O.* Structured I/O builds on our architecture's data model and lets applications persistently store tuples in environments. Comparable to different tables in a relational database, an environment's tuple storage is separate from that of other environments, and each tuple is stored in a specific environment. However, as discussed in Section 5.2.1, replication can be used to synchronize the contents of several environments. Comparable to the primary key in a relational database table, a tuple's ID uniquely identifies the tuple stored within an environment. In other words, at most one tuple with a given ID can be stored in a given environment. The structured I/O operations support the writing, reading, and deleting of tuples and are summarized in Table IV. They are atomic, so that their effects are predictable and can optionally use transactions to group several operations into one atomic unit. To use structured I/O, an application *binds* to an environment's tuple storage and then performs operations on the bound resource. In other words, the application specifies the environment whose tuple storage it wants to access (naming

the environment as described in Section 4.1) and then issues requests to the corresponding event handler returned by *one.world*'s kernel. All bindings are controlled by leases [Gray and Cheriton 1989], which limit the time an application can access the environment's tuple storage. Applications can renew these leases to increase the length of access or cancel them to relinquish access.

In the spirit of Unix's unified interface to storage and networking [McKusick et al. 1996], structured I/O also provides the same basic API for reading and writing tuples across the network. Because standard communication protocols, such as TCP, provide no persistence and employ only limited buffering, structured I/O networking supports only a subset of the operations shown in Table IV. In particular, it only supports the *put*, *read*, and *listen* operations and not transactions. To use structured I/O networking, applications bind to network endpoints instead of tuple storage. Network endpoints can be either UDP or TCP unicast sockets or UDP multicast sockets. Just as bindings for tuple storage, bindings for network endpoints are leased.

We chose to base I/O on a structured data model instead of using unstructured bytestrings because, by definition, tuples preserve the structure of application data and thus simplify the sharing and searching of data. Furthermore, tuples free applications from explicitly marshaling and unmarshaling data during I/O and from implementing their own, internal database functionality, which is a common strategy for desktop applications [Microsoft Corporation 1999] and leads to considerable duplication of effort between applications from different vendors. We chose tuples instead of XML [Bray et al. 1998] because tuples are simpler and easier to use. The structure of XML-based data is less constrained and also more complicated, including tags, attributes, and name spaces. Furthermore, interfaces to access XML-based data, such as DOM [Le Hors et al. 2000], are relatively complex.

Structured I/O distinguishes between storage and networking, instead of providing a unified tuple space service [Carriero and Gelernter 1986; Davies et al. 1998; Freeman et al. 1999; Murphy et al. 2001; Wyckoff et al. 1998], because such a separation better reflects how pervasive applications store and communicate data. On one hand, many applications need to modify stored data. For example, a personal information manager needs to update stored contacts and appointments. Structured I/O storage lets applications overwrite stored tuples by writing a tuple with the same ID as the stored tuple. In contrast, tuple spaces only support the addition, but not modification, of tuples. On the other hand, some applications, such as streaming media, need to directly communicate data in a timely fashion. Structured I/O networking provides that functionality. In constrast, tuple spaces store all tuples before delivering them and consequently retain them in storage, which is problematic for streaming media, as data tends to be large. As a result, tuple spaces represent a semantic mismatch for many pervasive applications, providing too little and too much functionality at the same time.

An additional concern is that tuple spaces are not amenable to layering in asynchronous systems. In particular, the *in* or *take* operation—an atomic read and delete—makes it hard to layer additional services, such as replication, on top of a tuple space. The problem is that the tuple to be deleted is only known

```
public interface EventHandler {
  // Handle the event.
  public void handle(Event e);
}
```

Fig. 8. The event handler interface. An event handler has a single method that takes the event to be processed as its only argument and returns no result.

after the *in* or *take* has been performed by the tuple space service, thus requiring that the replication layer intercept both the request and the corresponding response. In contrast, a replication layer on top of structured I/O only needs to intercept requests, never responses, because requests for the destructive *put* and *delete* operations are sufficiently descriptive to specify the affected tuples.

4.3 Execution Model

Having described *one.world*'s facilities for data management, we now turn to our architecture's execution model. In *one.world*, all functionality is implemented by event handlers that process asynchronous events. Events are appropriate for pervasive applications, as they make changes in an application's execution context—such as a person or device moving to a different location—explicit and provide the application with an opportunity to adapt to those changes. Since events are data, they too are represented by tuples. In addition to the ID and metadata fields common to all tuples, all events have a source field referencing an event handler. This event handler receives notification of failure conditions during event delivery and processing, as well as the response for request/response interactions. Furthermore, all events have a closure field, which can be of any allowable tuple field type including a tuple and is declared to be an `Object`. When responding to an event, by sending another event to the original event's source event handler, the closure of the original event is returned with the new event. Closures thus help simplify the implementation of event handlers, as applications can include any additional state needed for processing a response in the closure of the original request.

As shown in Figure 8, event handlers implement a uniform interface with a single method that takes the event to be processed as its only argument and returns no result. Any result for a request/response interaction must be sent as a regular event to the event handler referenced by the request's source field. Event delivery has at-most-once semantics, both for local and remote event handling. For remote event handling, at-most-once semantics are appropriate because, in lieu of transactional delivery protocols (which are too heavy-weight for basic event delivery), a remote device may fail after it has accepted an event but before the intended recipient had an opportunity to process it. For local event handling, exactly-once delivery is the norm. However, at-most-once semantics allow *one.world*'s implementation to recover from pathological overload conditions by selectively shedding load.

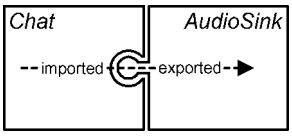

Fig. 9. Illustration of the relationship between imported and exported event handlers. Boxes represent components, indentations imported event handlers, and protrusions exported event handlers. The dotted arrow indicates the direction of event flow. In the example, the *Chat* component imports an event handler named *imported*, and the *AudioSink* component exports an event handler named *exported*. The two event handlers are linked, with *Chat* sending received audio messages to *AudioSink*, which then plays back the audio contained in the messages.

To simplify code reuse, application functionality is implemented by components. Components are units of code that support a uniform linking protocol and interact solely by exchanging events. They import and export event handlers, exposing the event handlers for linking, and are instantiated within specific environments. Although imported and exported event handlers can be added and removed after component creation, they are typically declared in a component's constructor. Imported and exported event handlers can be linked and unlinked at any time. After linking an imported event handler to an exported event handler, events sent to the imported event handler are processed by the exported event handler. Unlinking breaks this connection again. This relationship between imported and exported event handlers is illustrated in Figure 9.

An application's main component has a static initialization method that instantiates its components and performs the initial linking. It is called by our architecture when loading the application into its environment through the *load* operation listed in Table III. While the application is running, it can instantiate additional components, add and remove imported and exported event handlers, and relink and unlink components as needed. An example initialization method is shown in Figure 10. It instantiates a single component, representing Emcee's main component, and then performs two linking operations. After these simple initialization steps, the Emcee application is fully instantiated and ready to execute.

When an event is sent between components in different environments, the invocation of the exported, that is, receiving, event handler on the sent event is performed asynchronously. The corresponding invocation of the imported, that is, sending, event handler returns (almost) immediately. However, when an event is sent between components in the same environment, the event handler invocation is performed as a direct method call, so that the event is delivered reliably and efficiently. This default can be overridden at link-time, so that events within the same environment are also sent asynchronously.

We chose to use asynchronous events instead of synchronous invocations through, for example, regular procedure calls or a mix between regular procedure calls and asynchronous callbacks for three reasons. First and foremost, asynchronous events provide a natural fit for pervasive computing, as applications often need to raise or react to events, such as sending or receiving a text message or adapting to the execution context after a migration. Second,

```
public static void init(Environment env, Object closure) {
  // Create Emcee's component.
  Emcee comp = new Emcee(env);

  // Link the component with its environment.
  env.link("main", "main", comp);
  comp.link("request", "request", env);
}
```

Fig. 10. Code example for initializing an application. An initialization method takes as its arguments the environment for the application and a closure, which can be used to pass additional arguments, for example, from a command line shell. The example method first instantiates the Emcee component and then links that component with its environment, connecting Emcee's exported `main` event handler with the environment's corresponding imported event handler and Emcee's imported `request` event handler with the environment's corresponding exported event handler. The role of the `main` and `request` event handlers is explained in Section 4.3.1. Note that linked event handlers need not have the same name, although they do in this example. This code example is taken from Emcee's source code.

where threads implicitly store execution state in registers and on stacks, events make the execution state explicit. Systems can thus directly access execution state, which is useful for implementing features such as event prioritization or checkpointing and migration. Finally, taking a cue from other research projects [Chou et al. 1999; Gribble et al. 2000; Hill et al. 2000; Pai et al. 1999; Welsh et al. 2001] that have successfully used asynchronous events at very different points of the device space, we believe that asynchronous events scale better across different classes of devices than threads.

We chose a uniform event handling interface because it greatly simplifies composition and interposition. Event handlers need to implement only a single method that takes as its sole argument the event to be processed. Events, in turn, have a well-defined structure and are self-describing, making dynamic inspection feasible. As a result, event handlers can easily be composed with each other. For instance, the uniform event handling interface enables a flexible component model, which supports the linking of any imported event handler to any exported event handler. At the same time, the uniform event handling interface does not prevent the expression of typing constraints. When components declare the event handlers they import and export, they can optionally specify the types of events sent to imported event handlers and processed by exported event handlers.

4.3.1 *Interacting with the Kernel and Other Environments.* To an application, its environment appears to be a regular component. Each environment imports an event handler called *main*, which must be linked to an application's main component before the application can be started. It is used by *one.world* to notify the application of important events, such as activation, restoration, migration, or termination of the environment, and thus exposes contextual change to the application.

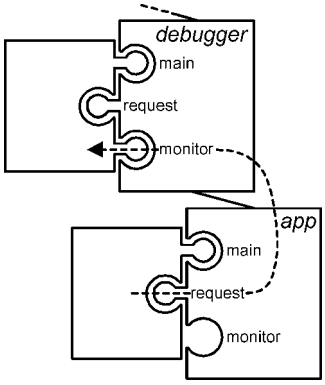

Fig. 11. Illustration of the request/monitor mechanism. Boxes on the left represent application components and boxes on the right represent environments. The dotted arrow indicates the direction of event flow. The *app* environment is nested within the *debugger* environment. The *debugger* environment's monitor handler is linked and thus intercepts all events sent to the *app* environment's request handler.

Each environment also exports an event handler called *request* and imports an event handler called *monitor*. Events sent to an environment's request handler are delivered to the first ancestral environment whose monitor handler is linked. The root environment's monitor handler is always linked to *one.world*'s kernel, which processes requests for environment operations, structured I/O, discovery, and remote event passing. Consequently, the environment request handler provides our architecture's system call interface, and applications use it for interacting with the kernel. For example, Emcee's initialization method as shown in Figure 10 links to its environment's request handler so that Emcee can utilize *one.world*'s services. Furthermore, by linking to the monitor handler, an application can interpose on all events sent to a descendant's request handler. For example, a debugger can monitor any application simply by nesting the application in its environment, linking to its own monitor handler, and observing the application's request stream. Similarly, a replication service can synchronize any application's data with another device by intercepting the application's structured I/O operations (in fact, as described in Section 5.2.1, our replication service does exactly that). This use of the *request/monitor mechanism* is illustrated in Figure 11.

We chose to represent environments as regular components, because it offers considerable flexibility and power. In particular, the request/monitor mechanism makes interposition trivial and greatly simplifies dynamic composition as illustrated above. Furthermore, because of the uniform event handler interface, the request/monitor mechanism is extensible; it can handle new event types without requiring any changes. Applications can inspect events using the tuple accessor methods shown in Figure 7, or pass them unexamined up the environment hierarchy. Finally, the same mechanism can be used to provide security by interposing a reference monitor [Anderson 1972] and auditing by logging an application's request stream. It thus obviates the need for fixing a particular security mechanism or policy in *one.world*'s kernel [Grimm and Bershad 2001].

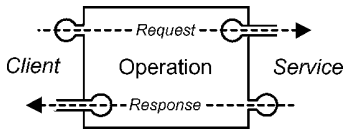

Fig. 12. Illustration of the operation library. The box represents an operation. Protrusions represent event handlers, which, unlike the event handlers shown in Figure 9 and Figure 11 are directly referenced. The dotted arrows indicate the direction of event flow. To manage asynchronous request/response interactions between a client and a service, an operation is interposed between the event handler accepting requests and the event handler expecting responses. The operation matches each request with exactly one response. It automatically detects timeouts and performs retries.

4.3.2 *Reliably Managing Event Exchanges.* While asynchronous events provide a good fit for pervasive computing, they also raise the question of how to manage event exchanges, especially when compared to the more familiar thread-based programming model. Of particular concern are how to maintain the state associated with pending request/response interactions and how to detect failures, notably lost events. These issues are especially pressing for pervasive computing environments, where people and devices keep coming and going, and where failures are a common, not an exceptional, occurrence. However, in our experience, established styles of event-based programming, such as event loops or state machines, are unsuitable. Since they combine all application logic into a single control block, they are only manageable for simple applications that process few distinct events. Furthermore, they do not help with detecting failures, including lost events.

After some experimentation, we found the following approach, which we call the *logic/operation pattern*, considerably more successful. Under this pattern, an application is partitioned into logic and operations, which are implemented by separate sets of event handlers. Logic are computations that do not fail, barring catastrophic failures, such as creating and filling in a text message. Operations are interactions that may fail, such as sending a text or audio message to its intended recipients. Operations maintain the state associated with these request/response interactions and also include all necessary failure detection and recovery code. A failure condition is reflected to the appropriate logic only if recovery fails repeatedly or the failure condition cannot be recovered from in a general way. As a result, the logic/operation pattern, unlike transparent, synchronous invocations, cleanly separates actual application logic from outside interactions, notably I/O, and their failure detection and recovery. Furthermore, unlike event loops or state machines, the logic/operation pattern does allow for the nesting of logic and operations, thus supporting more modular application code and scaling better with application size.

The `Operation` library reifies the logic/operation pattern. As illustrated in Figure 12, it is an event handler that connects an event handler accepting requests with an event handler expecting responses. For every request sent to an operation, the operation keeps the state of the pending interaction, including the request's closure, and sends exactly one response to the event handler expecting responses. The operation automatically detects timeouts and performs retries. If all retries fail, it notifies the event handler expecting responses of

```
operation = new Operation(0, Constants.OPERATION_TIMEOUT,
                          timer, request, continuation);
```

Fig. 13. Code example for creating an operation. The newly created operation does not perform any retries, times out after the default timeout, and uses the timer `timer`. Requests are sent to the `request` event handler and responses are forwarded to the `continuation` event handler. This code example is taken from Emcee's source code.

the failure through an exceptional event. Operations can be nested and can also be used on both sides of multi-round interactions, such as those found in many network protocols. As a result, operations provide an effective way for expressing complex interactions and structuring event-based applications.

To utilize the operation library, an application simply creates a new operation and then uses the operation instead of the original event handler for issuing requests. Example code for creating an operation is shown in Figure 13. The newly created operation connects the `request` event handler for receiving requests to the `continuation` event handler for receiving the corresponding responses. As shown in Figure 14 and Figure 15, the operation is then used for issuing requests instead of sending them to the `request` event handler directly.

4.4 Communication Model

The primary challenge in designing the communications facilities for *one.world* is to provide services that are more flexible than established point-to-point communications technologies and support a rich set of communication patterns. In particular, as people and devices move through the physical world, service discovery assumes a critical role for pervasive applications. After all, if an application cannot locate necessary resources in an ever changing computing environment, it cannot function. However, previous discovery systems, such as Jini [Arnold et al. 1999] and INS [Adjie-Winoto et al. 1999], expose considerably different APIs with distinct options, thus raising the question of what options to support in *one.world*'s discovery service.

To this end, we classify the major discovery options through a set of choices. The first choice reflects the *binding time* and determines when to perform a discovery query. With early binding, an application first uses discovery to resolve a query and then point-to-point communications to interact with the resolved resource. Early binding is appropriate when an application needs to repeatedly send events to the same resource, such as a specific, close-by wall display, or when services can be expected to remain in the same location, such as those running on dedicated servers. In contrast, late binding [Adjie-Winoto et al. 1999] combines query resolution and event routing into a single operation, and the discovery service routes the event directly to the matching resource. While late binding introduces a performance overhead for every sent event, it also is the most responsive and thus most reliable form of communication in a highly dynamic computing environment. The second choice reflects the *specificity* and determines the number of resources receiving an event. Anycast sends the event

to a single matching resource, such as the above mentioned wall display, while multicast sends the event to all matching resources, such as all users to chat with.

Taken together, the binding time and specificity cover the design space of previous discovery systems, where services make resources available under descriptors and clients query for matching resources. When determining which of these options to use in an application, the primary choice is whether to rely on early or late binding; whether to use anycast or multicast typically follows directly from the application's requirements. In our experience, late binding is generally preferable over early binding for pervasive applications, as it is more responsive in an ever changing computing environment. However, if an application sends many, possibly large messages to the same receiver in short succession, the overhead of repeatedly resolving discovery queries becomes noticeable, and early binding represents the more appropriate choice. At the same time, with early binding, the application needs to be prepared to rediscover the receiver if its computing context changes.

An additional, third choice reflects the *query target* and determines the entity on which to perform a discovery query. Typically, a query is performed on the resource descriptors, and the first two choices assume resource descriptors as query targets. However, the query can also be performed on the events themselves. In this case, an application receives all events sent through late binding discovery that match the query. Using the event as a query target constitutes a form of reverse lookup and is useful for implementing utilities that, for example, log and debug remote communications or bridge between different communication protocols by intercepting messages of one protocol and issuing those of another. In the former example, the reverse lookups are observing, that is, they do not count as matches for anycasts, while in the latter example, the reverse lookups are consuming, that is, they count as matches for anycasts (after all, the intent is to intercept events). To correctly intercept events, consuming reverse lookups need to be performed before regular, forward lookups.

We leverage our architecture's uniform data model and event handling interface to expose a common communications API to the discovery and remote event passing (REP) services. Our API supports all discovery options just described as well as point-to-point communications with only three operations, namely *export*, *resolve*, and *send*. In short, the *export* operation makes an event handler accessible across the network, while the *resolve* operation performs early binding discovery lookups, and the *send* operation routes events both for point-to-point communications and late binding discovery. Discovery resolves queries in a directory that represents all discoverable resources on the local network, while REP routes events directly to the specified device.

In detail, the three operations work as following:

Export. The *export* operation makes an event handler accessible across the network by establishing a mapping between a descriptor and the actual event handler; for discovery, all mappings are collected in a single directory for the local network. As shown in Table V, the descriptor's type determines how the event handler is exported. A `null` or a `Name` descriptor is used for point-to-point

```
SymbolicHandler destination;
if (null == fetchLocation) {
  // Location is unknown; use discovery.
  destination = new DiscoveredResource(new
    Query(new Query("",
                    Query.COMPARE_HAS_SUBTYPE,
                    UserDescriptor.class),
          Query.BINARY_AND,
          new Query("user", Query.COMPARE_EQUAL, fetchUser)));
} else {
  // Location is known; use point-to-point communications.
  destination = new
    NamedResource(fetchLocation, "/User/" + fetchUser);
}
operation.handle(new RemoteEvent(this, closure, destination, msg));
```

Fig. 14. Code example for sending a remote event. This example sends the event `msg` for user `fetchUser`, whose location `fetchLocation` may or may not be known. If the location is not known, the event is sent through late binding discovery. The discovery query matches tuples of type `UserDescriptor` whose `user` field equals `fetchUser`. If the location is known, the event is sent through point-to-point communications. The operation forwards the `RemoteEvent` to *one.world*'s kernel, which then performs the actual *send* operation. This code example is taken from Emcee's source code.

Table V. Options for Exporting Event Handlers to Remote Event Passing and Discovery

Descriptor	Explanation
`null`	Make the event handler accessible through point-to-point communications. The event handler can be referenced by the exporting device and the GUID returned by the export operation.
`Name`	Make the event handler accessible through point-to-point communications. The event handler can be referenced by the exporting device and the name contained in the `Name` tuple.
`Query`	Make the event handler accessible for reverse discovery lookups. An additional flag specifies whether the reverse lookups are consuming or observing. The former count as matches for anycast, while the latter do not. The event handler cannot be directly referenced. However, events sent through late binding discovery and matching the query are routed to the exported event handler.
All other tuples	Make the event handler accessible for regular discovery lookups. The event handler can be referenced by a query matching the specified tuple.

Descriptor specifies the tuple under which an event handler is exported. *Explanation* describes how the event handler can be accessed.

communications, while a `Query` or any other tuple is used for reverse and forward discovery lookups, respectively. REP provides two alternatives, so that clients can either reference a specific service instance (through a GUID) or a service independent of the current instance (through a name). Furthermore, when exporting an event handler to discovery, the event handler is also exported under a GUID, so that the *resolve* operation can make the event handler available for point-to-point communications. Comparable to the use of leases for

structured I/O (as described in Section 4.2.2), the resulting binding between the event handler and descriptor is leased.

Resolve. The *resolve* operation looks up event handlers in the discovery directory, so that they can be used for point-to-point communications. It takes a query and returns either any or all matching event handlers that have been previously exported for regular discovery lookups. If no event handler matches the query, the *resolve* operation results in a failure notification.

Send. The *send* operation sends an event to a previously exported event handler. The targeted event handler is specified by a so-called *symbolic handler* that contains the information necessary for routing the event. For late binding discovery, the symbolic handler specifies the query and whether to perform anycast or multicast. The event is delivered to any or all event handlers matching the query in the discovery directory. For point-to-point communications, the symbolic handler specifies the device exporting the event handler and the corresponding GUID or name, and the event is delivered to the event handler that has been exported under the specified GUID or name on the specified device. Both for discovery and point-to-point communications, if no actual event handler matches the symbolic handler, the sender is notified of the failure condition.

Example code for sending an event through both late binding discovery and REP is shown in Figure 14. It illustrates how an application can easily switch between late binding discovery and point-to-point communications, simply by using a different symbolic handler. Switching from anycast to multicast for late binding discovery is even simpler, as it requires only an additional Boolean argument for the constructor of the `DiscoveredResource`.

4.5 Application Persistence

one.world's checkpointing and migration services help to protect pervasive applications against major failures, such as a portable device's batteries running out, and to move them between devices, so that they can easily follow a person as she moves through the physical world. The checkpointing service provides the *checkpoint* and *restore* operations listed in Table III. The *checkpoint* operation captures the in-memory state of an environment tree and then stores the captured state as a tuple in the root of the tree. The *restore* operation reads a previously stored checkpoint and restores the execution state to the saved state. The migration service provides the *move* and *copy* operations listed in Table III. Both operations capture the in-memory state of an environment tree, move the tree, including the just created checkpoint and all stored tuples, to a different device, and then restore the checkpoint. They differ in that the *move* operation deletes the original environment tree, while the *copy* operation leaves the original tree intact. In contrast to transparent migration systems, such as Sprite [Douglis and Ousterhout 1991], our architecture's checkpointing and migration services are fully visible to applications. Notably, applications are explicitly notified after they have been restored from a checkpoint or have been migrated to a different device, so that they can adapt to a changed execution context.

Furthermore, if a checkpointing or migration operation cannot be completed, because, for example, the new device does not have sufficient resources to host an environment tree, the operation is aborted and the requesting application is notified of the error condition.

As hinted at by this first description, the functionality of the checkpointing and migration services can be defined more precisely in terms of three procedures: *capture-state()* to create a checkpoint tuple representing an environment tree's in-memory state, *transfer-tree()* to communicate an environment tree's complete contents from one device to another, and *restore-state()* to recreate an environment tree's in-memory state from a previously created checkpoint tuple. Using these three procedures, the *checkpoint* operation simply invokes *capture-state()* and stores the resulting checkpoint tuple in the root of the environment tree, while the *restore* operation reads such a checkpoint tuple and then invokes *restore-state()* on the tuple. Both the *move* and *copy* operations represent a sequence of *capture-state()*, *transfer-tree()*, and *restore-state()* invocations, with the difference that the *move* operation also destroys the original environment tree. We now discuss the three procedures in detail.

The *capture-state()* procedure creates a bytestring representing an environment tree's in-memory state. It relies on the virtual machine to provide a uniform execution platform across different hardware architectures and on object serialization to convert between virtual machine objects and bytestrings. By traversing all objects reachable from a set of well-defined roots (the *main* and *monitor* event handlers introduced in Section 4.3.1), the *capture-state()* procedure captures the in-memory state of the components instantiated in the environment tree. Since all communications in *one.world* are through asynchronous events, the *capture-state()* procedure also captures the environment tree's execution state by serializing pending ⟨*event handler, event*⟩ invocations. Comparable to bus stops in Emerald [Steensgaard and Jul 1995], which define application states that are safe to migrate, execution state can only be captured for pending ⟨*event handler, event*⟩ invocations. Invocations that are currently being executed need to run to completion; invocations that do not complete within a constant waiting period are forcibly terminated. The *capture-state()* procedure does not capture the state of currently executing ⟨*event handler, event*⟩ invocations, because capturing them requires access to the virtual machine's execution stack. However, many virtual machines, such as the Java virtual machine [Lindholm and Yellin 1999] but unlike the Squeak virtual machine [Guzdial and Rose 2002], do not explicitly expose their execution stacks and would thus require modifications, which would limit portability.

While the *capture-state()* procedure does capture the state of the environment tree's application objects and pending ⟨*event handler, event*⟩ invocations, it does *not* include references to resources outside the environment tree. Since environments are isolated from each other, only references to event handlers can be exchanged between environments; all other data is copied. Consequently, the *capture-state()* procedure tests each event handler to determine whether it is implemented by code running in one of the environments in the tree. If the event handler is part of the tree, it is written to the checkpoint. If it is not part of the tree, it is replaced by a `null` value. Environments thus provide

a well-defined boundary for the state included in a checkpoint, and nulling out event handlers provides a simple contract for revoking access to outside resources. The *capture-state()* procedure revokes access to outside resources in order to avoid residual dependencies [Powell and Miller 1983], which require an altogether well-connected computing environment. However, pervasive computing environments, with their reliance on wireless networking technologies such as 802.11 [Gast 2002] or Bluetooth [Bluetooth SIG 2002], often exhibit weaker connectivity than traditional local networks [Mummert et al. 1995; Terry et al. 1995]. Furthermore, disconnected operation is a relatively frequent occurrence, for example, as people travel in cars or on airplanes, and connections, such as those using cell phones, often have high latency and low bandwidth.

The *transfer-tree()* procedure eagerly communicates an environment tree and all its contents, including the checkpointed in-memory state and all persistently stored tuples, from one device to another in one atomic operation. It is eager, again, because of the weaker connectivity typically found in pervasive computing environments. For a *move* operation, the *transfer-tree()* procedure invalidates references from the outside into the original environment tree to expose the change in location. When sending an event to such a reference, the event is not transparently redirected through a forwarding address [Fowler 1985]; instead, the sender is notified that the resource has been moved. Invalidating references from the outside into the tree is unnecessary for a *copy* operation, as the original tree remains in place. However, because the original tree remains in place, the *transfer-tree()* procedure assigns fresh GUIDs to the environments being communicated through a *copy* operation, thus avoiding duplicates.

The *restore-state()* procedure recreates an environment tree's in-memory state from a checkpoint tuple simply by deserializing it. It then notifies all environments in the tree that they have been restored, moved, or copied. This notification is delivered to each environment's *main* event handler before any other ⟨*event handler, event*⟩ invocation can be performed, which gives code running in the affected environments an opportunity to reconnect to outside resources before resuming regular event processing. Because the restored environment tree's execution context has likely changed, discovery becomes a central service for reconnecting to outside resources, and, as discussed in Section 4.4, *one.world*'s discovery service has been carefully designed to expose an easy-to-program and flexible interface. Furthermore, we believe that explicitly restoring access to outside resources does not place an additional burden on developers, as applications running on our architecture already need to explicitly acquire resources at other points in their life cycles, such as when they are activated.

One important issue in providing checkpointing and migration is how to control the use of the two services. This issue is especially pressing for migration, as potentially untrusted applications move from one device to another. We rely on environment nesting, which gives an outer environment complete control over all nested environments, to address this issue. On the sending side, an outer environment can use the request/monitor mechanism to intercept a request to be migrated (that has been issued by a nested environment) and either modify

```
operation.handle(new
  EnvironmentEvent(null, this, EnvironmentEvent.CHECK_POINT,
                   env.getId()));
```

```
operation.handle(new RestoreRequest(null, this, env.getId(), -1));
```

```
operation.handle(new
  MoveRequest(null, user, user.env.getId(),
              "sio://"+location+"/User", false));
```

Fig. 15. Code examples for checkpointing, restoring, and moving an environment. The first code snippet checkpoints a user's environment `env`, the second restores the latest checkpoint, and the third moves a user's environment `user.env` to the device named `location`. For all snippets, the operation forwards the event to *one.world*'s kernel, which then performs the requested operation. Note that the first argument to each event's constructor is the source for that event and is automatically filled in by the operation. The code snippets are taken from Emcee's source code.

it or reject it. Similarly, on the receiving side, the future outer environments are notified by the *transfer-tree()* procedure that an environment is about to be migrated to this device, and they can modify the parent environment or reject the migration altogether. Environment nesting thus provides an effective mechanism for limiting how untrusted applications migrate across a network.

Environment nesting also enables a useful pattern for initiating checkpointing and migration. Under this pattern, the logic to decide when to checkpoint and restore an application or when and where to migrate an application is factored into its own environment. As a result, the checkpointing or migration logic can be reused across different applications, thus simplifying the development of pervasive applications. In fact, this pattern is used by Emcee, *one.world*'s Finder-like application management utility: As illustrated by the example code in Figure 15, it leverages the environment nesting to trivially checkpoint, restore, and move all of a user's applications.

Overall, *one.world*'s checkpointing and migration services leverage our architecture's other services as much as possible to avoid complexity and to provide a clean and useful model for their operation. In particular, they rely on the virtual machine to provide a uniform execution environment across different devices and hardware architectures. They rely on environments to clearly delineate what state to capture and what state not to capture. They also rely on environments for controlling migration, both on the sending and the receiving side, and for factoring the checkpointing and migration logic out of pervasive applications. Furthermore, they rely on the integration of tuple storage with environments to save checkpoints with an application and to migrate an application's persistent data with itself. Next, they rely on asynchronous events to make an application's execution state explicit. They also rely on asynchronous events to notify an application of a completed *restore*, *move*, or *copy* operation,

thus exposing the application's changed execution context. Finally, they rely on discovery so that an application can easily adapt to a changed execution context by restoring access to the appropriate resources. In other words, by building on the other elements of *one.world*'s programming model, our architecture's checkpointing and migration services can provide important functionality without incurring undue complexity.

5. EXPERIMENTAL EVALUATION

In this section, we present the user-space services, utilities, and applications we and others have built and present the results of our experimental evaluation, which is based on these programs. The goal is to answer the question of whether *one.world* is good enough for building pervasive applications. Or, to be more consistent with the approach discussed in Section 2, we are trying to determine whether focusing on the requirements of pervasive computing has resulted in a system architecture that enables developers to effectively build adaptable applications. However, both questions are rather general and hard to answer. Accordingly, we rely on four, more specific criteria and corresponding questions to evaluate our architecture:

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 are especially interested in how making applications adaptable impacts programmer productivity.

Performance. Is system performance acceptable? This criterion determines if our architecture performs well enough to support actual application workloads. Since our goal is to make applications adaptable, we are especially interested in whether applications respond quickly to changes in their runtime context.

Utility. Have we enabled others to be successful? This criterion determines whether others can build real pervasive applications on top of *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.

After a short overview of *one.world*'s implementation (see Grimm [2000] for a more in-depth description), we address these four criteria, with one criterion per subsection. We discuss the user-space programs we and others built for our architecture along the way, presenting our own programs in Section 5.2 and the Labscape application in Section 5.5. In addition to these primary programs, students have also built a music sharing system, a messaging system for future, intelligent home appliances, a graphical debugger, and a web server on top of our architecture. Since the students' experiences largely confirm the results derived from our own programs and the Labscape digital laboratory assistant, we do not discuss them in this article.

To summarize the results, we show that *one.world* (1) is sufficiently complete to support interesting programs on top of it, (2) is not significantly harder to program than with conventional programming styles, (3) has acceptable performance, with applications reacting quickly to changes in their runtime context, and, most importantly, (4) enables others to successfully build pervasive applications. In other words, our experimental results show that *one.world* does, in fact, enable developers to build applications that adapt to change instead of forcing users to constantly reconfigure their systems and, consequently, they validate our approach.

5.1 Overview of *one.world*'s Implementation

As described in detail in Grimm [2000], *one.world*'s implementation runs on Windows and Linux PCs. It is largely written in Java, which provides us with a safe and portable execution platform. We use a small, native library to generate GUIDs, as they cannot be correctly generated in pure Java. Furthermore, we use the Berkeley DB [Olson et al. 1999], which provides a transactional hash table on top of an unreliable file system, to implement tuple storage. Our implementation currently lacks support for transactions as part of structured I/O (though, the individual structured I/O operations are fully implemented) and for loading code from environments. As a result, applications cannot use transactions to group several structured I/O operations into a single, atomic unit, and application code must be manually distributed across all devices that are expected to run an application. We expect that adding these features to our implementation will be straightforward. Our implementation does, however, include library support for building GUI-based applications, for a command line shell, and for converting between files and stored tuples (to enable the exchange of data between conventional operating systems and *one.world*).

one.world has been released as an open source package. The implementation has approximately 19,000 non-commenting source statements (NCSS). Our entire source tree, including regression tests, benchmarks, and applications, has approximately 40,000 NCSS or 109,000 lines of well-documented code, representing an overall development effort of about six man years. A Java archive file with the binaries for *one.world* itself is 514 KB. The GUID generation library requires 28 KB on Windows and 14 KB on Linux systems, while the Berkeley DB libraries require another 500 KB on Windows and 791 KB on Linux systems.

Our implementation does not rely on features that are unique to Java. It requires a type-safe execution environment to provide basic protection. Next, the ability to customize the code loading process (through classloaders [Liang and Bracha 1998] in our Java-based implementation) is used to isolate different environments from each other and to ensure that only trusted code can access abstractions, such as files or sockets, that are outside *one.world*'s programming model. Furthermore, application control over threads is used to implement asynchronous events. Finally, reflection and object serialization are used to implement tuple-based I/O, checkpointing, and migration as well as to

automatically copy events passed between environments. As a result, *one.world* could also be implemented on other platforms that provide these features, such as Microsoft's Common Language Runtime [Thai and Lam 2002].

5.2 Completeness

To evaluate our architecture, including determining completeness, we built a set of user-space programs. In this section, we first describe their functionality and implementations and highlight how they utilize *one.world*'s services. In particular, we present a replication service in Section 5.2.1, followed by Emcee—our user and application management utility, as well as Chat—our text and audio messaging application, in Section 5.2.2. We then discuss the results regarding completeness in Section 5.2.3.

5.2.1 *Replication Service.* To provide ubiquitous access to people's information, pervasive applications need to access the corresponding data items, even if several people share the same data and access it from different and possibly disconnected devices. One strategy for providing this capability is to replicate the data. Our replication service does just that and makes stored tuples accessible on multiple devices that may be disconnected. By providing replication as a common application building block, our replication service simplifies the development of pervasive applications, as developers need not reimplement this important, but also complex capability.

Our replication service is patterned after Gray et al.'s [1996] two-tier replication model. A master node owns all data and replicas have copies of that data. Replicas can either be connected or disconnected. In connected mode, updates are final and performed directly on the master. In disconnected mode, updates are tentative and logged on the replica. When a replica becomes connected again, it synchronizes with the master by replaying its log against the master and by receiving updates from the master. The replica may then disconnect again or continue in connected mode.

We chose two-tier replication over Bayou's epidemic replication model [Petersen et al. 1997; Terry et al. 1995] for two reasons. First, two-tier replication is easier to explain to users, as tentative updates may only change once, during synchronization, and not repeatedly. Relying on an easy-to-explain model for replication is important, because pervasive computing is expressly targeted at supporting all people and not just computer experts. Second, on a more technical level, two-tier replication avoids system delusion [Gray et al. 1996]. Delusion occurs when large numbers of replicas repeatedly reconcile with each other in the absence of a master and consequently diverge further and further from each other.

The implementation of our replication service runs in user-space and, as illustrated in Figure 16, exploits environment nesting—through the request/monitor mechanism—to interpose on an application's access to tuple storage. The replicator logs updates in the log environment when in disconnected mode and forwards them to the master when in connected mode. On reconnection of a disconnected device, instead of sending individual updates as remote

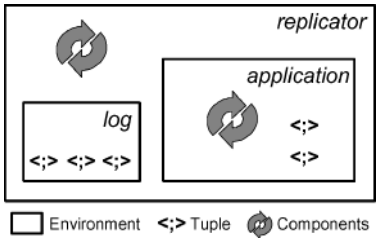

Fig. 16. Illustration of our replication service's structure. The *replicator* environment intercepts all storage operations issued by the application. In disconnected mode, the replicator logs updates in the *log* environment. In connected mode, it directly forwards them to the master.

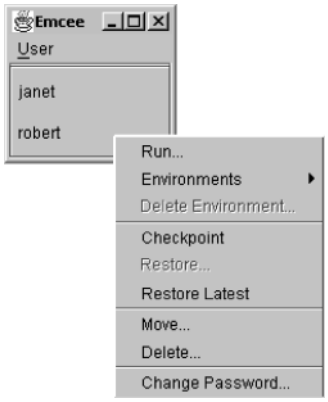

Fig. 17. A screenshot of Emcee's user interface. The main window lists the users whose applications run on the device. A popup menu for each user, shown for the user named *robert*, is used to perform most operations, such as running a new application or checkpointing a user's applications. The user menu supports the creation of new users and the fetching of a user's applications from another device.

events, the log is sent to the master in one operation by copying the log environment. Similarly, updates are sent from the master to the replica by migrating an environment containing such updates.

As illustrated by our replication service, migration can serve as an internal building block for applications and can be used to simplify communications. Furthermore, because environments host both computations and data, migration provides an effective way to move application-specific reconciliation logic to the master: the replicator simply instantiates the necessary components in the log environment before copying it. Finally, our replication service is not limited to using migration internally; rather, the master and its replicas are migratable themselves. Migrating the master is useful when, for example, upgrading the computer the master is running on; migrating a replica is useful when the user is switching devices.

5.2.2 *Emcee and Chat.* Emcee, whose user interface is shown in Figure 17, manages users and their applications. It includes support for creating new users, running applications for a user, and checkpointing all of a user's

Fig. 18. A screenshot of Chat's user interface. The user interface is divided into four panels, which can be independently expanded or collapsed by checking or unchecking the corresponding checkbox. The *listen* panel shows received text messages and provides volume control for audio playback. The *send message* panel lets the user send text messages, and the *send audio* panel lets the user send audio, either from a microphone or from stored audio tuples. Finally, the *subscribe* panel lets the user select the channels she is currently listening to.

applications. Emcee also provides the ability to move or copy applications between users, simply by dragging an application's flag icon, as shown in the upper right corner of Figure 18, and dropping it onto a user's name in the main window. Finally, it supports moving all of a user's applications between devices, so that the applications can follow a user as she moves through the physical world. Applications can either be pushed from the current device to another device, or they can be pulled from another device to the current device. Emcee can manage any *one.world* application; an application does not need to implement any features specific to Emcee. However, to support drag and drop through the flag icon, an application's developer needs to add three lines of code to the application.

As illustrated in Figure 5, the implementation of Emcee structures the environment hierarchy according to the pattern `/User/<user>/<application>`. Emcee runs in the `/User` environment and uses a child environment for each user and a grandchild for each application. Each user's root environment stores that user's preferences, including her password, and application checkpoints. Emcee's main event handler, which is linked to the `/User` environment's main event handler as shown in Figure 10, processes *one.world* events for activation and termination. On activation, Emcee creates its user interface and then starts performing user-requested operations. On termination, it tears down the user

interface and then stops all event processing. The implementation of most operations is straight-forward, since, as illustrated in Figure 15, they directly map to *one.world*'s primitives.

The exception is fetching a user's applications from a remote device, which uses a challenge-response protocol to authenticate the user to the remote instance of Emcee currently hosting the user's applications. After the user has been successfully authenticated, the remote Emcee initiates a migration of the user's environment tree to the requesting device. As illustrated in Figure 14, the initial remote event for this fetcher protocol is routed through late binding discovery if the user has not specified her applications' location. Otherwise, it is sent directly to the remote device. To process this initial remote event, Emcee exports the corresponding event handler twice for each user: once to REP under the `/User/<name>` name and once to discovery under a `UserDescriptor` tuple whose `user` field equals the user's name.

Chat, whose user interface is shown in Figure 18, provides text and audio messaging. It is based on a simple model, under which users send text and audio messages to a channel and subscribe to a channel to see and hear the messages sent to it. The implementation sends all messages through late binding discovery, using TCP as the underlying transport protocol for text messages and UDP for audio messages. For each subscribed channel, Chat exports an event handler to discovery, which then receives the corresponding messages. Audio can either be streamed from a microphone or from sound tuples stored in an environment. Since music files tend to be large, they are converted into a sequence of audio tuples when they are imported into *one.world*. Using the tuple IDs as symbolic references, the sequence of audio tuples forms a doubly-linked list. As Chat is streaming audio messages, it traverses this list and reads individual tuples on demand, buffering one second of audio data in memory.

Emcee and Chat illustrate the power of migration combined with dynamic composition through discovery and environment nesting. Discovery connects applications in the face of migration. Because Chat uses late binding discovery to route text and audio messages, messages are correctly delivered to all subscribed users even if the users switch devices. At the same time, environment nesting makes it possible to easily migrate applications, such as Chat, that have no migration logic of their own. Emcee controls the location of a user's applications simply by nesting the applications in its environment. Chat does not need its own migration logic and can automatically benefit from future improvements in Emcee's migration support, such as using smart badges to identify a user's location instead of requiring the user to explicitly move and fetch applications.

5.2.3 *Discussion.* The applications presented in this section clearly show that our architecture is powerful enough to support a variety of useful programs. Furthermore, our replication service and Emcee, both of which help providing ubiquitous access to a person's data and applications, demonstrate that services and utilities can indeed be implemented in user-space. The key feature for enabling user-space services and utilities is *one.world*'s environment hierarchy. Nested environments make it easy to control other programs and, through the

request/monitor mechanism, to interpose on their request streams. However, as discussed in Section 4.2.2, we had to carefully design the interface for structured I/O to support layering, an effort akin to previous work on stackable file systems [Heidemann and Popek 1994].

Furthermore, in the course of developing the above programs, we did encounter one relatively minor limitation to *one.world*'s APIs: the performance evaluation of our replication service [Grimm et al. 2001] suggests that the durability guarantees of structured I/O storage can result in too high an overhead for some applications. In particular, immediately forcing each *put* operation to disk is unnecessary when logging updates in disconnected mode, because all updates are already tentative. We thus designed (but have not yet implemented) a simple extension to structured I/O, under which applications can optionally request that the destructive *put* and *delete* operations provide only relaxed durability guarantees and are lazily written to disk. Just as with traditional file system interfaces, applications using this option need to explicitly perform a *flush* operation to force pending updates to disk.

5.3 Complexity

To evaluate the effort involved in writing adaptable applications, we analyzed the process of implementing Emcee and Chat. The general theme for developing Emcee and Chat was that "no application is an island." Consistent with a computing environment where people and devices keep coming and going, applications need to assume that their runtime context changes quite frequently and that external resources are not static. Furthermore, they need to assume that their runtime context may be changed by other applications. These assumptions have a subtle but noticeable effect on the implementations of Emcee and Chat. Rather than asserting complete control over the environments nested in the `/User` environment, Emcee dynamically scans its children every second and updates the list of users in its main window accordingly. Similarly, it scans a user's environments before displaying the corresponding popup menu (which is displayed by selecting the "Environments" menu entry shown in Figure 17.

For Chat, these assumptions show up throughout the implementation, with Chat verifying that its internal configuration state is consistent with its runtime context. Most importantly, Chat verifies that the user, that is, the parent environment's name, is still the same after activation, restoration from a checkpoint, and migration. If the user has changed, it updates the user name displayed in its title bar, adjusts default channel subscriptions, and clears its history of text messages. Furthermore, it runs without audio if it cannot initialize the audio subsystem, but retains the corresponding configuration state so that it can resume playback when migrating to a different device. Finally, it also silences a channel if the audio tuples have been deleted from their environment.

In our experience with Chat and Emcee, programming for change has been tractable. The implementation aspects presented above are important for Emcee's and Chat's correct operation, but are not overly complex. Furthermore, programming for change can also simplify an application's implementation. For example, when Emcee fetches a user's applications, it needs some way to

Table VI. Breakdown of Development Times in Hours for Emcee and Chat

Activity	Time (hrs.)
Learning Java APIs	21.0
User interface	47.5
Logic	123.5
Refactoring	6.0
Debugging and profiling	58.0
Total time	256.0

The times shown are the result of three developers implementing the two applications over a three month period. The activities are discussed in the text.

detect that the user's applications have arrived on the local device. But, because Emcee already scans its children every second, the arrival will be automatically detected during a scan and no additional mechanism is necessary. To put it differently, the initial effort in implementing an adaptable mechanism—dynamically scanning environments—has more than paid off by simplifying the implementation of an additional application feature—fetching a user's applications.

To quantify the effort involved in building Emcee and Chat, we tracked the time spent developing the two programs. They were implemented by three developers over a three month period. During that time, we also added new features to *one.world*'s implementation and debugged and profiled the architecture. Overall, implementing Emcee and Chat took 256 hours; a breakdown of this overall time is shown in Table VI. *Learning Java APIs* is the time spent for learning how to use Java platform APIs, notably the JavaSound API utilized by Chat. *User interface* is the time spent for implementing Emcee's and Chat's GUI. *Logic* is the time spent for implementing the actual application functionality. *Refactoring* is the time spent for transitioning both applications to newly added *one.world* support for building GUI-based applications. It does not include the time spent for implementing that support in our architecture, as that code is reusable (and has been reused) by other applications. Finally, *debugging and profiling* is the time spent for finding and fixing bugs in the two applications and for tuning their performance.

Since Emcee and Chat have 4,231 non-commenting source statements (NCSS), our overall productivity is 16.5 NCSS/hour.[2] As discussed above, *one.world* is effective at making programming for change tractable. In fact, adding audio messaging, not reacting to changes in the runtime context, represented the biggest challenge during the implementation effort, in part because we first had to learn how to use the JavaSound API. We spent 125 hours for approximately 1750 NCSS, resulting in an approximate productivity of 14 NCSS/hour. If we subtract the time spent learning Java platform APIs (including the JavaSound API), working around bugs in the Java platform, and

[2]Productivity is traditionally measured in lines of code per hour or LOC/hour. NCSS/hour differs from LOC/hour in that it is more exact and ignores, for example, a brace appearing on a line by itself. As a result, NCSS/hour can be treated as a conservative approximation for LOC/hour.

refactoring our implementation from the total time, our overall productivity increases to 20.4 NCSS/hour, which represents an optimistic estimate of future productivity. Our actual productivity of 16.5 NCSS/hour lies at the lower end of the results reported for considerably smaller and simpler projects [Prechelt 2000], but is almost twice as large as the long-term results reported for a commercial company [Ferguson et al. 1999]. Based on this, we extrapolate that programming for change does not decrease overall programmer productivity and conclude that it is not significantly harder than more conventional programming styles.

5.4 Performance

To determine whether our implementation performs well enough for real application usage, we measured the scalability of migration and late binding discovery. Migration and discovery are the two services the programs discussed in this section rely on the most and, in general, are indispensable for realizing applications that follow people through the physical world. Furthermore, to characterize system and application reactivity, we explored how Chat reacts to changes in its runtime context. Reactivity is especially important for pervasive applications, as they need to continuously adapt to changes in their runtime context. It also marks a clear point of departure from traditional distributed applications such as Microsoft's Outlook, which locks up for minutes when it cannot reach the corresponding Exchange server.

All measurements reported on in this section were performed using Dell Dimension 4100 PCs, with Pentium III 800 MHz processors, 256 MB of RAM, and 45 or 60 GB 7,200 RPM Ultra ATA/100 disks. The PCs are connected by a 100 Mb switched Ethernet. We use Sun's HotSpot client virtual machine 1.3.1 running under Windows 2000 and Sleepycat's Berkeley DB 3.2.9.

To quantify the scalability of migration, we conducted a set of micro-benchmarks. For the micro-benchmarks, we use a small application that moves itself across a set of devices in a tight loop. We measure the application circling 25 times around three PCs for each experiment. To test the scalability of migration under different loads, we add an increasing number of tuples carrying 100 bytes of data, tuples carrying 100,000 bytes of data, and copies of our Chat application in separate sets of experiments.

The results show that migration latency increases linearly with the number of stored tuples or copies of Chat. We measure a throughput of 12.6 KB/second for tuples carrying 100 bytes of data, 16.2 KB/second for copies of Chat, and 1,557 KB/second for tuples carrying 100,000 bytes of data. In the best case (tuples carrying 100,000 bytes), migration utilizes 12% of the theoretically available bandwidth and is limited by how fast stored tuples can be moved from one PC to the other. Since moving a stored tuple requires reading the tuple from disk, sending it across the network, writing it to disk, and confirming its arrival, a better performing migration protocol than the currently implemented one should optimistically stream tuples and thus overlap the individual steps instead of moving one tuple per protocol round (as the current implementation does).

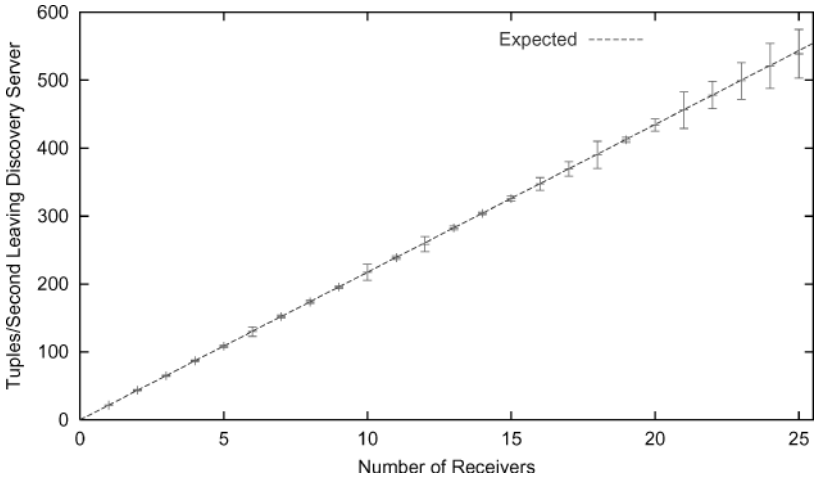

Fig. 19. Discovery server throughput under an increasing number of receivers. Throughput is measured as the number of audio messages leaving the discovery server. The results shown are the average of 30 measurements, with error bars indicating the standard deviation. Each audio message carries 8 KB of audio data.

To quantify the scalability of late binding discovery, we stream audio messages between a varying number of Chat applications. We chose to measure streaming audio, because messages are large (see below) and must be delivered on time, thus exercising our implementation of the discovery service. As described in detail in Grimm [2000], the implementation of our discovery service relies on a centralized server, which holds the discovery directory and, to ensure availability, is automatically elected from all devices running *one.world* on the local network. Elections are called aggressively and complete within a fixed time period. Discovery clients tolerate any resulting inconsistencies by exporting bindings between descriptors and the corresponding event handlers to all visible servers while forwarding requests to only one server.

Figures 19 and 20 show the discovery server throughput under an increasing number of receivers for a single sender and an increasing number of senders for a single receiver, respectively. Throughput is measured as audio messages leaving the discovery server, and the results shown are the average of 30 measurements. Each audio message carries 8 KB of uncompressed audio data at CD sampling rate, which corresponds to 10,118 bytes on the wire when forwarding from the sending node to the discovery server and 9,829 bytes when forwarding from the discovery server to the receiving node. The difference in on-the-wire sizes stems largely from the fact that messages forwarded to the discovery server contain the late binding query, while messages forwarded from the discovery server do not. The receivers and senders respectively run on four PCs; we use Emcee's support for copying applications via drag and drop to spawn new ones.

When increasing the number of receivers, discovery server throughput increases almost linearly with the number of receivers. However, when increasing the number of senders, discovery server throughput levels off at about

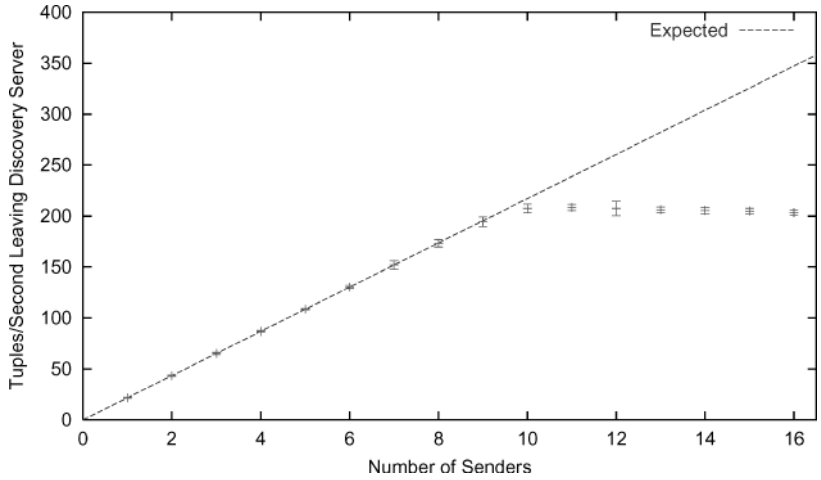

Fig. 20. Discovery server throughput under an increasing number of senders. As in Figure 19, throughput is measured as the number of audio messages leaving the discovery server. The results shown are the average of 30 measurements, with error bars indicating the standard deviation. Each audio message carries 8 KB of audio data.

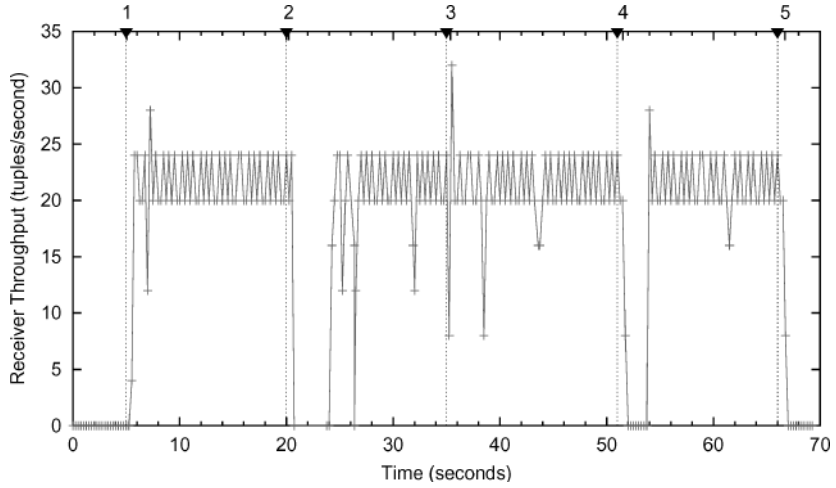

Fig. 21. Audio messages received by Chat in a changing runtime environment. Chat is subscribed to an audio channel at point 1. It is then moved to a different node at point 2. The node hosting the discovery server is shut down gracefully at point 3 and forcibly crashed at point 4. The audio channel is unsubscribed at point 5.

10 senders and slightly degrades thereafter. At 10 senders, the PC running the discovery server becomes CPU bound. While the cost of processing discovery queries remains low, the cost of processing UDP packets and serializing and deserializing audio messages comes to dominate that PC's performance.

Figure 21 illustrates system and application reactivity by showing the audio messages received by Chat as its runtime context changes. As for the discovery server throughput experiments, each audio message carries 8 KB of

uncompressed audio data at CD sampling rate. Unlike the migration latency experiments, Chat is managed by Emcee and runs within its user's environment. At point 1, Chat is subscribed to an audio channel and starts receiving audio messages shortly thereafter. At point 2, Chat is moved to a different device and does not receive audio messages for 3.7 seconds while it migrates, reinitializes audio, and reregisters with discovery. After it has been migrated and its receiving event handler has been reexported to discovery, it starts receiving audio messages again. The PC running the discovery server is gracefully shut down at point 3. *one.world* proactively calls a discovery server election, and the stream of audio messages is not interrupted. By contrast, at point 4, the PC running the discovery server is forcibly crashed. The stream of audio messages is interrupted for 2.3 seconds until a new discovery server is elected and Chat's receiving event handler is forwarded to the new discovery server. This period is governed by detecting the crashed discovery server, which requires two missed server heartbeats or 2 seconds. Finally, at point 5, Chat is unsubscribed from the audio channel and stops receiving audio messages shortly thereafter.

Overall, our performance evaluation shows that service interruptions due to migration or forced discovery server elections last only a few seconds, which compares favorably with Microsoft's Outlook hanging for several minutes. Furthermore, while migration latency generally depends on the number and size of stored tuples, it takes only 7 seconds for an environment storing 8 MB of audio data, which is fast enough when compared to a person moving through the physical world. Finally, our architecture performs well enough to support several independent streams of uncompressed audio data at CD sampling rate. However, our evaluation also suggests that discovery server scalability is limited. Adding a secondary discovery server could improve the scalability of our discovery service and would also eliminate service interruption due to forced server elections. Since, as described in Grimm [2000], our implementation already works with more than one discovery server, this change is a relatively simple one.

5.5 Utility

To determine utility, we supported the Labscape project in porting their digital laboratory assistant introduced in Section 2.3 to our architecture. Remember that Labscape's goal is to seamlessly capture, organize, and present biology processes in order to help biologists perform reproducible experiments. The Labscape application tries to achieve this goal by having an experimental guide follow a researcher from touchscreen to touchscreen as she moves through the laboratory. The constraints are that (1) the Labscape developers are programmers and not system builders, and (2) the resulting application has to be good enough to be used by real biologists every day. In other words, the Labscape application has to be responsive, stable, and robust. The application needs to react quickly to changes in its execution context, and it needs to be continuously available. Furthermore, when a failure occurs, its effects should be localized and it should be easy to recover.

The Labscape team actually created three different implementations of their digital laboratory assistant. The first version centralizes all processing and relies on remote windowing through VNC [Richardson et al. 1998] to display the individual guides on a laboratory's touchscreens. In the Labscape team's experience, the first version is neither responsive nor robust. While a different remote windowing system, such as X Windows [Nye 1995], might have alleviated the performance concerns, it would not have eliminated the central point of failure. Furthermore, the first version's reliance on remote windowing precludes more advanced features, such as a researcher reviewing her work while commuting homewards and being disconnected from the digital laboratory.

The second version of Labscape uses distributed processing; code and data follow the researchers through the laboratory by migrating from touchscreen to touchscreen. The Labscape team implemented the second version directly in Java, using TCP sockets for communications and their own, application-specific migration layer to move the guides. In their experience, the second version is neither stable nor robust, thus prompting the Labscape team to port their application to *one.world*. This third version of Labscape has the same basic structure as the second version. However, it relies on our architecture's late binding discovery for communications and the migration service for moving guides between touchscreens. The resulting application is responsive, stable, and robust.

The structure of the second and third versions is illustrated in Figure 22. The individual application services work as following. The *device access service* collects experimental data from RFID and barcode scanners and location updates from IR sensors. It converts the data and the updates into the appropriate events and then forwards them to the *proximity service*, which tracks researchers' locations. For experimental data, the proximity service determines the researcher that performed the scanning operation and, in turn, forwards the data to the researcher's guide. For location updates, the proximity service updates its internal data structures and then advises the researcher's guide to move to the closest touchscreen. The *WebDAV service* is used to publish experimental data on the World Wide Web. Finally, the *state service* serves as the final repository for all experimental data, which it receives from the researchers' guides.

Porting to *one.world* resulted in three major benefits over the Java version. First, it reduced the development time from nine to four man months. In part, the reduced development time stems from the fact that the Labscape team did not have to redesign their application and could reuse existing code, as our architecture remains neutral on an application's structuring (as stipulated in Section 3). Second, porting simplified code maintenance and improved performance. In the Java version, every major modification of the guide requires corresponding changes in the migration layer. Yet, despite the application-specific migration layer, moving a guide in the Java version is five to ten times slower than in the *one.world* version. In Arnstein et al. [2002], the Labscape team reports that migration latencies for the *one.world* version are between 2.5 seconds for moving a guide with no experimental data and 7.1 seconds for moving a guide with 64 samples, representing a large experiment. These results are

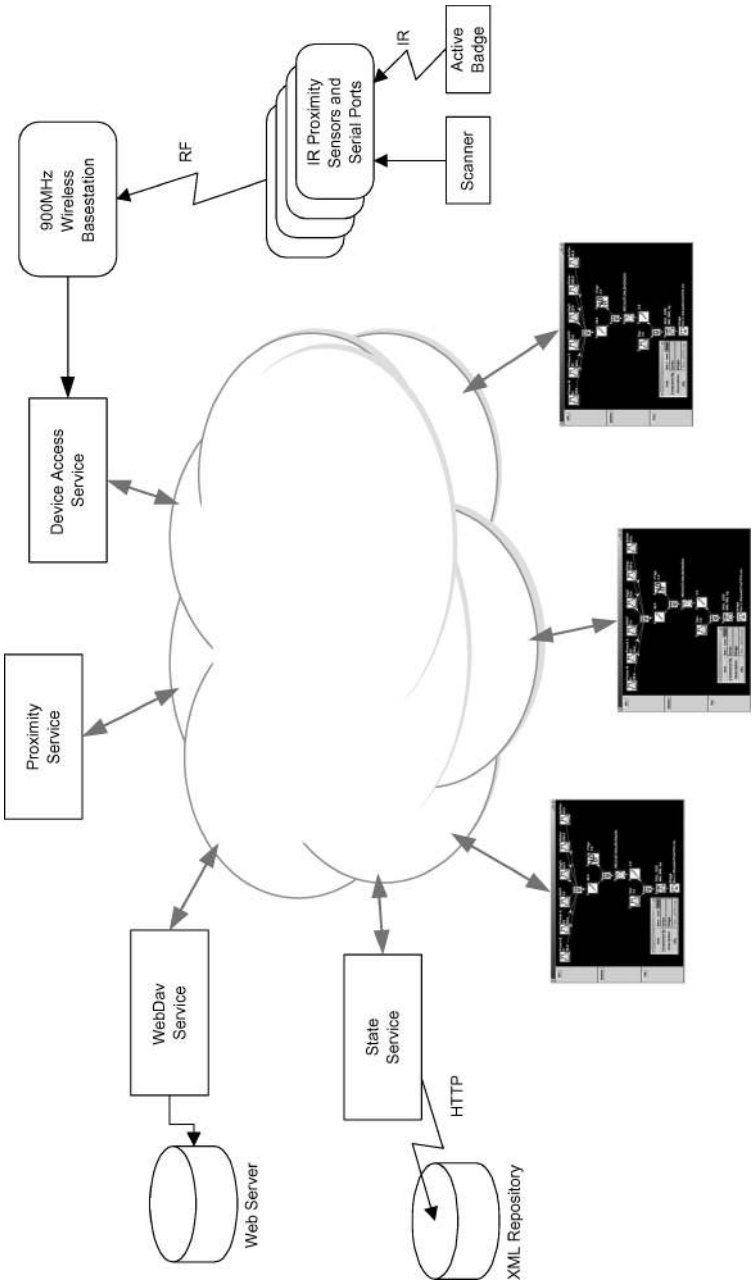

Fig. 22. Illustration of Labscape's structure (second and third version). The experimental guides (shown at the bottom) use migration to follow researchers as they move through the laboratory and communicate with a set of application services. The second version is implemented in plain Java and relies on an application-specific migration layer as well as TCP sockets. In contrast, the third version is implemented on top of *one.world* and uses our architecture's migration service and late binding discovery. The individual application services are explained in the text.

consistent with our own measurements, as reported above, and are acceptable when compared to a researcher moving through the laboratory—while migration latencies around a minute are not.

Finally, porting considerably improved application uptime and resilience to failures. The Java version has a mean time between failures (MTBF) of 30 minutes, compared to an order of days for the *one.world* version. The short MTBF of the Java version stems from a lack of appropriate system support as well as from buggy application code. Porting to *one.world* can eliminate the first cause but not the second cause. At the same time, system support can help with graceful failure recovery. In particular, after a failure of the Java version, the *entire* digital laboratory has to be restarted. In contrast, by building on late binding discovery instead of direct TCP connections, the *one.world* version allows for a piecemeal restarting of application services and thus is considerably more resilient in the face of buggy application code.

Once more, the Labscape application illustrates the power of combining *one.world*'s migration with late binding discovery. As programs move from one device to another, they easily communicate with each other by routing events through the discovery service. At the same time, Labscape's use of migration differs from the other programs discussed in this section. Unlike the replication service—which uses migration as an internal building block—and Emcee—which controls how other applications are migrated, the guides in Labscape migrate themselves. Given this versatility of migration, we believe it to be a general building block for system services, utilities, and applications alike.

To improve their application's resilience to failures, the Labscape team plans to further exploit *one.world*'s features. In particular, they intend to replicate the proximity service, which currently represents a single point of failure. The corresponding code changes are simple, as they only require changing event delivery from anycast to multicast and ignoring duplicate events in the guides. Furthermore, the Labscape team plans to add support for disconnected operation to the guides. Currently, the guides require that the state service be continuously available, so that they can directly forward updates. Support for disconnected operation can easily be added by logging pending updates in local tuple storage and forwarding them once the state service becomes available again.

Overall, the Labscape application demonstrates that *one.world* provides a solid platform for building and running real pervasive applications. However, the Labscape team did encounter three limitations of our architecture. First, in the Labscape team's experience, *one.world* events are harder to program than, for example, Java Swing events. In particular, they would like to write more concise event code and see better support for managing asynchronous interactions (in addition to our operation library). Second, *one.world* has its own data model based on tuples and its own network communications in the form of REP and discovery. As a result, it is unnecessarily hard to interact with legacy and web systems. We revisit both issues in more detail in the next section.

Finally, due to security constraints enforced by our architecture, the Labscape team had difficulties in reusing existing, third-party Java libraries. More

specifically, our architecture prevents applications from accessing Java's `java.lang.System` class and makes select methods, notably `arraycopy()` to copy the contents of arrays and `getProperty()` to access system properties, accessible through its own `one.world.util.SystemUtilities` class. Using a different class to access these methods does not represent a restriction for applications written from scratch; developers 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*. To address this issue, we developed a simple utility that, through binary rewriting, transforms existing libraries and replaces invocations to `System`'s methods with the corresponding *one.world* methods.

6. DISCUSSION

As shown in the previous section, the user-space programs we and others built provide us with a solid basis for evaluating *one.world*'s design and implementation. The process of developing and using these programs also helped us gain a better understanding of the strengths and limitations of our architecture and its implementation. In this section, we focus on the resulting insights and identify lessons that are applicable beyond our work as well as opportunities for future research into system support for pervasive applications.

The user-space programs presented in Section 5 make extensive use of *one.world*'s services and illustrate the power of a design that follows the three requirements of change, ad hoc composition, and pervasive sharing:

Embrace contextual change. Event-based notification cleanly exposes change to applications. For example, the Labscape application uses events to expose location changes to a researcher's guide, enabling that guide to move to a close-by touchscreen. Furthermore, Chat relies on events to automatically adjust its configuration when the user owning the application changes.

Encourage ad hoc composition. Environment nesting and discovery make it easy to dynamically compose functionality. For example, Emcee relies on environment nesting to control a user's applications, and both our debugger and replication service use the request/monitor mechanism to interpose on an application's request stream. Furthermore, Emcee, Chat, and the Labscape application all rely on discovery to connect different application instances or services. Moreover, discovery not only simplifies communications in the face of migrating applications—as is the case for Chat—but also increases applications' resilience to failures—as illustrated by Labscape.

Recognize sharing as a default. Tuples simplify the capturing and searching of information. For example, the Labscape application directly encodes experimental data as tuples, thus allowing for the direct searching of that data. Furthermore, the separation of data—in the form of tuples, and functionality—in the form of components, provides considerable flexibility when compared to systems that combine data and functionality in objects. For instance, we can add music to a running Chat application, simply by importing the corresponding files into Chat's environment. Conversely, we can improve existing audio

capabilities by instantiating the corresponding components in Chat's environment. Yet, while upgrading the application, we do not need to change stored audio tuples.

Additionally, migration and remote event passing provide powerful primitives that cover the spectrum between collocation and remote interaction. On one side, we rely on migration to make a user's applications available on a close-by device. On the other side, we rely on REP to let applications communicate with each other.

The central role played by environments in our architecture implies, in our opinion, a more general pattern, namely that *nesting is a powerful paradigm for controlling and composing applications*. To reiterate, nesting provides control, as illustrated by Emcee, and nesting can be used to extend applications, as illustrated by our replication service. Nesting thus makes it possible to easily factor important and possibly complex behaviors and provide them as common application building blocks. Furthermore, nesting is attractive because it preserves the relationships between nested environments. For instance, when audio tuples are stored in a child environment of Chat's environment, the environment with audio tuples remains a child, even if Chat's environment is nested in a user's environment and subsequently moved between devices.

While the user-space programs provide ample examples for the power of our architecture, they also helped in identifying several limitations. We discuss the issues raised by our data model in Section 6.1, followed by event processing in Section 6.2, leases in Section 6.3, and structured I/O's unified interface to storage and communications in Section 6.4. We then discuss user interfaces in Section 6.5 and, finally, the interaction between *one.world* and the outside world in Section 6.6.

6.1 Data Model

The biggest limitation of our architecture is that, to access a tuple, a component also needs to have access to the tuple's class. This does not typically pose a problem for applications, which have access to their own classes. However, it does pose a problem for services, such as discovery, that process many different types of data for many different applications. One solution, which we have not yet implemented, uses a generic tuple class, say `StaticTuple`, to provide access to the fields of different classes of tuples by using the accessor methods shown in Figure 7. When passing a tuple across protection domains or when sending it across the network, the system tries to locate the tuple's class. If the class can be accessed, the tuple is instantiated in its native format. If the class cannot be accessed, the tuple is instantiated as a `StaticTuple`. In contrast to the `DynamicTuple` described in Section 4.2, whose fields are dynamically added and removed as well as dynamically typed, a `StaticTuple` preserves all typing information of a tuple's original class. In particular, it ensures that field values conform with the fields' declared types, and it does not support the dynamic addition or removal of fields. This solution works because services that process many different types of data already use the accessor methods instead of accessing a tuple's fields directly.

A `StaticTuple` can provide access to a tuple's fields even if the tuple's class cannot be accessed. At the same time, it cannot capture the semantic constraints expressed by the tuple's `validate()` method or the human-readable representation expressed by the `toString()` method. As a result, it represents a workable yet incomplete solution. The fundamental problem is that we have taken a single-node programming methodology, namely 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*, which expresses schemas as data and not as code. With a data-centric data model, applications still need to access a data item's schema in order to manipulate the data item. However, since the schemas themselves are data and not code, they are easier to inspect programmatically and not tied to a specific execution platform. As a result, we conclude that *data-centric data models provide better interoperability than programmatic data models*.

We believe that defining an appropriate data-centric data model is an important topic for future research into pervasive computing. The challenge is to define a data model that meets conflicting requirements. On one side, to support the pervasive sharing of information, the data model must be general and supported by a wide range of platforms. One possible starting point is XML Schema [Biron and Malhotra 2001; Thompson et al. 2001]. It already defines the data model for SOAP [Box et al. 2000], which is the emerging standard for remote communications between web services and used, for example, by Microsoft's .NET platform [Thai and Lam 2002]. On the other side, 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 DOM [Le Hors et al. 2000], and providing a more efficient encoding, perhaps by using a binary encoding [Martin and Jano 1999] or by compressing the data [Liefke and Suciu 2000]. Ideally, a data-centric data model should be as easy to program as field access for tuples in our architecture. Probably, such a data model will specify a generic data container and a provision for automatically mapping data to application-specific objects, comparable to our proposed use of `StaticTuple`.

While tuples are limited by being based on a programmatic data model, the uniform and ubiquitous use of tuples in our architecture has proven to be very powerful. In particular, it allowed us to gracefully evolve the discovery service and integrate new functionality not found in other discovery systems. The initial design of our discovery service did not include support for reverse lookups (as described in Section 4.4). However, while implementing Chat, we needed some means for debugging remote communications through late binding discovery. We considered adding a dedicated interface for debugging discovery, but rejected that option as not general enough. We then converged on reverse lookups as a more flexible technique. Because of our architecture's uniform use of tuples, integrating reverse lookups with discovery was easy. Since events are tuples, they can be treated just like any other data, and reverse lookups on events can be directly expressed as regular queries. Furthermore, since queries are tuples, we simply added one more option to the *export* operation. We thus

conclude that *the uniform use of structured data enables new functionality and helps to gracefully evolve a system*.

6.2 Event Processing

We still believe that asynchronous events provide a good fit for pervasive applications, as they make changes in an application's execution context explicit. But even with library support for managing request/response interactions, in the form of our architecture's operation library, developers require additional facilities for writing event-based applications. In particular, several event handlers in the programs described in Section 5 need to process many different types of events or perform different actions for the same type of event depending on the event's closure. Their implementation requires large if-then-else blocks that use `instanceof` tests to dispatch on the type of event or more general tests to dispatch on the value of the closure. The result is that these event handlers are not very modular and are relatively hard to understand, modify, or extend—an issue expressly noted by the Labscape team. This suggests the need for better programming language support to structure event handlers. Alternatives include predicate dispatch as provided by JPred [Millstein 2004] or pattern matching as provided by Standard ML [Milner et al. 1997].

Overall, our experience with event-based programming suggests that, contrary to Ousterhout [1996], *asynchronous events are as hard to program as threads*. Just like threads, asynchronous events can result in complex interactions between components. For example, a better performing alternative to the migration protocol measured in Section 5.4 might optimistically stream tuples rather than waiting for an acknowledgement for each tuple. However, providing flow control for streamed events can easily replicate the full complexity of TCP's flow control [Stevens 1994]. Furthermore, just as a system can run out of space for new threads, event queues can run out of space for new events. Finally, asynchronous events are not a panacea and some interactions must be synchronous. For example, timers to detect lost events must be scheduled synchronously because scheduling them asynchronously would use the same mechanism whose failure they are meant to detect.

6.3 Leases

As described in Section 4.2 and Section 4.4, resource access in our architecture is leased. Leases provide an upper bound on the time resources can be accessed, although leases can still be revoked by *one.world*'s kernel before their expiration, notably when an application is migrated. To make the use of leases practical, we introduced a lease maintainer library early on in our implementation effort. The lease maintainer automatically renews the lease it manages until it is explicitly canceled. While lease maintainers work most of the time, they can still fail, allowing a lease to expire prematurely. For example, when a device is overloaded, lease renewal events may not be delivered on time. Furthermore, when a device, such as a laptop or handheld computer, is hibernating, renewal events cannot be delivered at all. As a result, applications need to be prepared to reacquire local resources, such as their environment's tuple storage, even

though the resources are guaranteed to be available. We thus conclude that *leases do not work well for controlling local resources*. Instead, we prefer a simple bind/release protocol, optionally with callbacks for the forced reclamation of resources, and use leases only for controlling remote resources.

6.4 A Unified Interface to Storage and Communications

As described in Section 4.2.2, we took a cue from Unix and carefully designed structured I/O to expose the same basic interface for storage and communications (though, in contrast to tuple spaces, structured I/O storage and networking are distinct services). However, *none* of the programs we and others built actually use structured I/O networking; they all rely on remote event passing and discovery for network communications. Only REP and discovery themselves employ structured I/O networking in their implementations. We believe that developers favor REP and discovery over structured I/O networking for remote communications because the former are higher-level and more flexible services. As a result, we conclude that we overdesigned structured I/O. We could have omitted structured I/O networking and instead used a simpler, internal networking layer for implementing REP and discovery. In other words, *storage and communications are orthogonal to each other and best implemented by separate services with distinct interfaces*.

6.5 User Interface

All GUI-based programs running on top of *one.world* use Java's Swing toolkit [Walrath and Campione 1999] to implement their user interfaces. The integration between Swing's event model and *one.world*'s event model has worked surprisingly well. When an application needs to react to a Swing event, it generates the corresponding *one.world* event and sends it to the appropriate event handler. Long-lasting operations, such as fetching a user's applications, are broken up into many different *one.world* events, which are processed by our architecture's thread pools [Gribble et al. 2000; Welsh et al. 2001]. Swing's event dispatching thread, which executes an application's user interface code, is only used for generating the first *one.world* event in a sequence of *one.world* events. As a result, applications in our architecture, unlike other applications using Swing, do not need to spawn separate threads for processing long-lasting operations. In the opposite direction, when an application needs to update the user interface in reaction to a *one.world* event, it simply schedules the update through Swing's `SwingUtilities.invokeLater()` facility.

An important limitation of Swing and other, comparable toolkits is that the user interface does not scale across different devices. For example, we successfully used Emcee and Chat on tablet computers but would be hard pressed to also run them on, say, handheld computers. However, an important property of pervasive computing environments is the variety of supported devices. While most of these devices rely on screens—albeit considerably smaller ones than those used with PCs—for output and some pointing device for input, some devices, such as Sony's Aibo robotic dog, employ entirely different forms of input and output, including speech. Consequently, we believe that an important topic for future research into pervasive computing is how to implement scalable user

interfaces. One promising approach, which is being explored by the user interface markup language [Abrams and Helms 2002] (UIML) and the Mozilla project's XML-based user interface language [Bullard et al. 2001] (XUL), is to define a declarative specification of an application's interface, which is automatically rendered according to a device's input and output capabilities.

An unexpected lesson relating to user interfaces is that *GUI-based applications help with the testing, debugging, and profiling of a system*. Once we started using Emcee and Chat, we quickly discovered several bugs in our architecture that we had not encountered before. The two applications also helped us with identifying several performance bottlenecks in our implementation. We believe that this advantage of GUI-based applications stems from the fact that GUIs encourage users to "play" with applications. As a result, the system is exercised in different and unexpected ways, especially when compared to highly structured regression tests and interaction with a command line shell. Furthermore, it is easier to run many GUI-based applications at the same time and, consequently, to push a system's limits.

6.6 Interacting with the Outside World

To provide its functionality, *one.world* prevents applications from using abstractions not defined by our architecture. By default, applications cannot spawn their own threads, access files, or bind to network sockets. These restrictions are implemented through a Java security policy [Gong 1999]. As a result, specific applications can be granted access to threads, files, and sockets by modifying a device's security policy. However, because these abstractions are not supported by our architecture, applications are fully responsible for their management, including their proper release when an application is migrated or terminated.

Access to sockets is especially important for applications that need to interact with the outside world, including with Internet services. For example, we have used a modified security policy to let a web server run in our architecture. The web server's implementation is split into a front end and a pluggable back end. The front end manages TCP connections, translates incoming HTTP requests into *one.world* events, and translates the resulting responses back to HTTP responses. It also translates between MIME data and tuples by relying on the same conversion framework used for translating between files and stored tuples. The default back end provides access to tuples stored in nested environments.

In the opposite direction, it is not currently practical for outside applications to communicate with *one.world* applications through REP or discovery, especially if the outside applications are not written in Java. Because of our programmatic data model, an outside application would have to reimplement large parts of Java's object serialization, which is unnecessarily complex. However, to provide ubiquitous information access, pervasive applications must easily interact with each other, independent of the underlying systems platform, as well as with Internet services. After all, the Internet is the most successful distributed system, used by millions of people every day. We believe that moving to a data-centric, XML-based data model, as discussed above, and using standardized

communication protocols will help in providing better interoperability between pervasive applications, even if they run on different system architectures, and with Internet services. To put it differently, *modern distributed systems need to be compatible with* Internet protocols *first and offer additional capabilities second*.

7. RELATED WORK

one.world incorporates several technologies that have been successfully used by other systems. The main difference is that our architecture integrates these technologies into a simple and comprehensive framework, with the goal of enabling applications to adapt to an ever changing computing environment. Furthermore, where necessary, our architecture does introduce new services. Most importantly, our environment service is unique in that it combines the management of computations and persistent storage into a single, hierarchical structure. Other innovations include our remote event passing and discovery services, which expose an integrated API that covers the spectrum of network communications options, our migration service, which makes migration in the wide area practical, and our operation library, which effectively manages asynchronous interactions. In this section, we highlight relevant systems and discuss their differences when compared to *one.world*. Note that we have already reviewed systems that adapt transparently in Section 2.2.

The environment service was inspired by the ambient calculus [Cardelli 1999]. Similar to environments, ambients are containers for data, functionality, and other ambients, resulting in a hierarchical structuring. Unlike environments, which are used to implement pervasive applications, ambients are abstractions in a formal calculus and are used to reason about mobile computations. The MobileSpaces agent system [Satoh 2000] also relies on a hierarchical structuring, where agents can be embedded within other agents. Like environments, MobileSpaces agents are migrated together with all nested agents. Unlike environments, MobileSpaces agents provide only limited isolation (an outer agent can directly access the objects of an inner agent), cannot interpose on the request stream of inner agents (as provided in *one.world* through the request/monitor mechanism), and do not include persistent storage.

Asynchronous events have been used across a wide spectrum of systems, including networked sensors [Hill et al. 2000], embedded systems [Chou et al. 1999], user interfaces [Petzold 1998; Walrath and Campione 1999], and large-scale servers [Gribble et al. 2000; Pai et al. 1999; Welsh et al. 2001]. Of these systems, *one.world*'s support for asynchronous events closely mirrors that of DDS [Gribble et al. 2000] and SEDA [Welsh et al. 2001]. As a result, it took one author a very short time to reimplement SEDA's thread pool controllers in *one.world*. Our architecture also provides two improvements over these two systems. First, in DDS and SEDA, the event-passing machinery is exposed to application developers, and events need to be explicitly enqueued in the appropriate event queues. In contrast, as described in Grimm [2000], *one.world* automates event-passing through the use of proxied event handlers. Second, DDS and SEDA lack support for structuring event-based applications beyond

breaking them into so-called stages (which map to environments in our architecture). While stages represent a significant advance when compared to prior event-based systems, operations in *one.world* provide additional structure for event-based applications and simplify the task of writing asynchronous code.

Odyssey [Noble et al. 1997] relies on asynchronous notifications to expose contextual change to applications. It is based on a client/server model, where applications' access to services is mediated by the Odyssey runtime. Under this model, applications specify allowable fidelity ranges for the services they use. The runtime, in turn, relies on type-specific components to map these fidelity ranges to actual resources, for example, to select an appropriate resolution for streaming video. When a service cannot be provided within the requested fidelity range, for example, because of insufficient network bandwidth, the Odyssey runtime notifies the application through an upcall, thus allowing the application to select a different fidelity range. Odyssey's use of asynchronous upcalls for exposing contextual change is comparable to our architecture's use of events. However, Odyssey has been designed as a minimal extension to a traditional operating system (NetBSD). As a result, it is far less flexible in specifying what resources to access (it only supports file names) and in notifying applications of contextual change (it only supports a single upcall with three simple parameters). Furthermore, it lacks more advanced services that help applications adapt, such as our architecture's migration and discovery services. At the same time, Odyssey's framework for mapping fidelity ranges to actual resources based on a resource's type is complimentary to our own work.

Starting with Linda [Carriero and Gelernter 1986], tuple spaces have been used to coordinate loosely coupled applications [Davies et al. 1998; Freeman et al. 1999; Murphy et al. 2001; Wyckoff et al. 1998]. Departing from the original tuple space model, several of these systems support more than one global tuple space and may even be extended through application-specific code, for example, to automatically synchronize a local and a remote tuple space. Our architecture's use of tuples differs from these systems in that, as discussed in Section 4.2.2, structured I/O storage is a separate service from communications—whether through structured I/O networking or through remote event passing and discovery—and more closely resembles a database interface than Linda's *in*, *out*, and *rd* operations. At the same time, applications that require a traditional tuple space can easily implement such a service on top of remote event passing and structured I/O storage. Linda's *out* and *rd* operations map directly to structured I/O's *put* and *read* operations (though, every tuple written through a *put* must have a fresh GUID as its ID). Linda's *in* operation can be implemented as a transactional *read* and *delete*.

Like tuple spaces, the information bus helps with coordinating loosely coupled services [Oki et al. 1993]. Unlike tuple spaces, it is based on a publish/subscribe paradigm and does not retain sent messages in storage. While its design is nominally object-based, data exchanged through the bus is self-describing and separate from service objects, comparable to the separation of data—in the form of tuples—and functionality—in the form of components—in *one.world*. The information bus dynamically matches senders and receivers based on so-called subjects. Subjects are hierarchically structured strings,

similar to DNS names, and matching supports equality testing as well as wildcards. Messages are published under specific subjects and then delivered to all interested parties. In its ability to deliver messages to receivers based on a property of the message, the information bus resembles our architecture's reverse discovery lookups. However, the information bus provides only a very limited form of reverse lookup and does not support forward lookups at all. Interestingly, the information bus also includes an option for point-to-point communications (albeit through synchronous remote method invocations), just like our architecture supports both point-to-point communications and the dynamic matching between senders and receivers.

On the surface, Sun's Jini [Arnold et al. 1999] appears to provide many of the same services as our architecture. However, Jini embodies a fundamentally different approach to building distributed applications: it extends single-node programming methodologies, is strongly object-oriented, and relies on remote method invocations. As a result, Jini requires an overall well-behaved computing environment, and its services are rather limited when compared to the corresponding services in *one.world*. In particular, Jini requires a statically configured discovery server. Moreover, Jini's discovery supports only early binding and simple equality queries. Furthermore, Jini does not provide isolation between applications running on the same Java virtual machine (JVM), thus making it impossible to terminate ill-behaved programs without terminating all programs running on that JVM. Likewise, Jini synchronously sends remote events through Java RMI, thus exposing the sender to arbitrary delays on the receiving side. Finally, Jini relies on distributed garbage collection [Plainfossé and Shapiro 1995] (DGC) for controlling objects' lifetimes. However, the illusion of a global pool of objects provided by DGC is misleading. Objects can still be prematurely reclaimed, for example, when devices are disconnected for a sufficiently long time and DGC's internal leases expire. DGC also makes it unnecessarily hard to provide migration on top of Jini. Since DGC controls objects' lifetimes, a migration service cannot move objects without either proxying every remotely accessible object or being fully integrated with DGC's implementation.

In addition to Jini, the intentional naming system [Adjie-Winoto et al. 1999] (INS), the secure discovery service [Czerwinski et al. 1999] (SDS), the service location protocol [Guttman et al. 1999] (SLP), and universal plug and play [Microsoft Corporation 2000] (UPnP) all provide the ability to locate resources by their descriptions. Out of these systems, INS comes the closest to our architecture's discovery service. Like *one.world*, INS supports early and late binding as well as anycast and multicast. Furthermore, comparable to the use of discovery server elections in our architecture, INS' servers automatically form an overlay network to route late binding messages; though, individual servers still need to be manually configured. SDS and SLP both explore how to secure service discovery. Additionally, SDS includes a mechanism for aggregating service descriptions into a global hierarchy of discovery servers. Both efforts are complimentary to our own work. Finally, UPnP is largely targeted at automatically connecting PCs and stand-alone devices, such as printers and displays. As a result, it supports only simple matching queries (comparable to subject matching for the information bus). At the same

time, UPnP does include support for event-based notifications when a device's state changes. The main difference between these services and our own is that *one.world* integrates discovery with point-to-point communications, resulting in a simple and elegant API that covers the spectrum of remote communications options. Furthermore, our discovery service is the only one to support reverse lookups.

A considerable number of projects have explored migration in distributed systems [Milojičić et al. 1999]. Notable examples include migration at the operating system level, as provided by Sprite [Douglis and Ousterhout 1991], and at the programming language level, as provided by Emerald [Jul et al. 1988; Steensgaard and Jul 1995]. In these systems, providing support for a uniform execution environment across all nodes and for transparent migration of application state has resulted in considerable complexity. In contrast, many mobile agent systems, such as IBM's aglets [Lange and Oshima 1998], avoid this complexity by implementing what we call "poor man's migration". They do not provide transparency and only migrate application state by serializing and deserializing an agent's objects. Since these systems are thread-based, they do not migrate an application's execution state, forcing application developers to implement their own mechanisms for managing execution state. Because of its programming model, *one.world* can strike a better balance between the complexity of fully featured migration and the limited utility of poor man's migration. While *one.world* does not provide transparency, it does migrate an application's execution state as well as its persistent data.

Several other projects are exploring aspects of system support for pervasive applications. Notably, InConcert, the architectural component of Microsoft's EasyLiving project [Brumitt et al. 2000], provides service composition in a dynamic environment by relying on location-independent names and asynchronous events. Furthermore, iROS, the operating system for Stanford University's iRoom project [Johanson et al. 2002], features an asynchronous event distribution system, a shared tuple space that not only stores but also transforms data, and an automatic user interface generation system. An important common theme to these efforts and our own is the need for networked communications that are asynchronous and dynamically match senders with receivers. iROS and *one.world* also share their reliance on tuples for representing all data, including events. At the same time, we fundamentally differ in our approaches. The EasyLiving and iRoom projects seek to better integrate the applications running in a single, intelligent room. As a result, they reuse existing applications wherever possible and provide only as much system support as strictly necessary. In contrast, *one.world* has been designed from the ground up to meet the requirements of pervasive applications. Consequently, our architecture is more complete and powerful, but also requires that applications be written from scratch.

Like *one.world*, Carnegie Mellon University's Aura project [Garlan et al. 2002; Satyanarayanan 2001] targets pervasive computing environments that are not limited to a single room. Unlike *one.world*, Aura takes a more backwards compatible approach: it builds on existing system support for mobile computing, including Coda [Kistler and Satyanarayanan 1992; Mummert et al. 1995] and

Odyssey [Noble et al. 1997], and supports standard desktop operating systems, such as Windows and Linux. To accommodate different operating systems, users' tasks in Aura are described as abstract services that are automatically mapped onto the underlying platform and its native applications. Task migration in Aura then builds on these descriptions to move platform-independent representations between devices. Another important component of Aura is the integration of staging servers to off-load resource intensive computations from mobile devices and to improve performance when accessing far-away data sources.

Several efforts, including Globe [van Steen et al. 1999], Globus [Foster and Kesselman 1997], and Legion [Lewis and Grimshaw 1996], explore an object-oriented programming model and infrastructure for wide area computing. They share the important goal of providing a common execution environment that is secure and scales across a global computing infrastructure. However, these systems are targeted at collaborative and scientific applications running on conventional PCs and more powerful computers. As a result, these systems are too heavy-weight and not adaptable enough for pervasive computing environments. Furthermore, as argued in Section 2, we believe that their reliance on RPC for remote communications and on objects to encapsulate data and functionality is ill-advised.

8. CONCLUSIONS

In this article, we have explored how to build pervasive applications. Based on the observation that many existing distributed systems extend single-node programming methodologies, with the result that users need to manually adapt their computing environment in the presence of change, we have suggested a more suitable approach. Under this approach, system support exposes distribution rather than hide it, so that applications can see contextual change and then adapt to it. More specifically, system support needs to address the following three requirements. First, systems need to embrace contextual change, so that applications can implement their own strategies for handling changes. Second, systems need to encourage ad hoc composition, so that applications can be dynamically connected and extended in an ever changing runtime environment. Third, systems need to recognize sharing as the default, so that applications can make information accessible anywhere and anytime.

We have presented *one.world*, a system architecture for pervasive computing, that represents a first stab at exploring how to realize this approach. Our architecture builds on four foundation services that directly address the three requirements. First, a virtual machine provides a uniform execution environment across all devices and supports the ad hoc composition between applications and devices. Second, tuples define a common type system for all applications and simplify the sharing of data. Third, events are used for all communications and make change explicit to applications. Finally, environments host applications, store persistent data, and—through nesting—facilitate the composition of applications and services. On top of these foundation services, our architecture provides a set of system services that address common application needs,

including discovery to locate resources across the network and migration to move or copy applications between devices.

We have validated our architecture by supporting the Labscape team in porting their digital biology laboratory assistant to our architecture and by developing our own programs—including a replication service, a user and application manager, and a text and audio messaging system. Our experimental evaluation has demonstrated that *one.world* (1) is sufficiently complete to support additional services, utilities, and applications on top of it, (2) is not significantly harder to program than with conventional programming styles, (3) has acceptable performance, with applications reacting quickly to change, and, most importantly, (4) enables others to successfully build pervasive applications. However, our experimental evaluation has also shown that the scalability of our implementation, notably that of service discovery, is limited, making it suitable only for pervasive computing environments with several dozens of people and devices. Yet, despite these performance concerns, our evaluation has demonstrated that *one.world* lets developers effectively build applications that adapt to change, thus validating our approach.

Based on our own and others' experiences with *one.world*, this article has also identified important lessons, both positive and negative, that are applicable beyond this work. Notably, we have demonstrated that nesting is a powerful paradigm for controlling and composing applications and that the uniform use of structured data enables new functionality and helps to gracefully evolve a system. However, we have also found that—unlike our architecture, which defines its own communication protocols—modern distributed systems need to be compatible with Internet protocols first and offer additional capabilities second. Furthermore, we have found that asynchronous events are as hard to program as threads, that leases do not work well for controlling local resources, and that storage and communications are orthogonal to each other and best implemented by separate services with distinct interfaces. We have also suggested areas for future work on pervasive computing, specifically data-centric data models and scalable user interfaces. More information on our architecture, including a source release, is available at `http://www.cs.nyu.edu/rgrimm/one.world/`.

ACKNOWLEDGMENTS

Ben Hendrickson implemented parts of structured I/O. We thank the members of the Labscape project, the students of University of Washington's CSE 490dp, Daniel Cheah, and Kaustubh Deshmukh for using *one.world* to build their applications. Mike Swift and the anonymous reviewers provided valuable feedback on earlier versions of this article.

REFERENCES

Abrams, M. and Helms, J. 2002. User interface markup language (UIML), version 3.0. Draft specification, Harmonia, Inc., Blacksburg VA (Feb.). Available at `http://www.uiml.org/specs/uiml3/DraftSpec.htm`.

Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., and Lilley, J. 1999. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. Kiawah Island Resort, SC. 186–201.

ANDERSON, J. P. 1972. Computer security technology planning study. Tech. Rep. ESD-TR-73-51, Vol. I, (Oct.). Electronic Systems Division, Air Force Systems Command, Bedford, MA. Also AD-758 206, National Technical Information Service.

ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. 1996. Serverless network file systems. *ACM Trans. Comput. Syst. 14*, 1 (Feb.), 41–79.

ARNOLD, K., O'SULLIVAN, B., SCHEIFLER, R. W., WALDO, J., AND WOLLRATH, A. 1999. *The Jini Specification*. Addison-Wesley.

ARNSTEIN, L., GRIMM, R., HUNG, C.-Y., KANG, J. H., LAMARCA, A., SIGURDSSON, S. B., SU, J., AND BORRIELLO, G. 2002. 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, 30–44.

BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C., AND EGGERS, S. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain, CO. 267–284.

BIRON, P. V. AND MALHOTRA, A. 2001. XML schema part 2: Datatypes. W3C recommendation, World Wide Web Consortium (May.). Cambridge, MA.

BIRRELL, A. D., LEVIN, R., NEEDHAM, R. M., AND SCHROEDER, M. D. 1982. Grapevine: An exercise in distributed computing. *Comm. ACM 25*, 4 (Apr.), 260–274.

BLUETOOTH SIG. 2002. Specification of the Bluetooth system. Specification Version 1.1 (Feb.). Bluetooth SIG.

BOX, D., EHNEBUSKE, D., KAKIVAYA, G., LAYMAN, A., MENDELSOHN, N., NIELSEN, H. F., THATTE, S., AND WINER, D. 2000. Simple object access protocol (SOAP) 1.1. W3C note, World Wide Web Consortium, (May.). Cambridge, MA.

BRAY, T., PAOLI, J., AND SPERBERG-MCQUEEN, C. M. 1998. Extensible markup language (XML) 1.0. W3C recommendation, World Wide Web Consortium, (Feb.). Cambridge, MA.

BREWER, E. A., KATZ, R. H., CHAWATHE, Y., GRIBBLE, S. D., HODES, T., NGUYEN, G., STEMM, M., HENDERSON, T., AMIR, E., BALAKRISHNAN, H., FOX, A., PADMANABHAN, V. N., AND SESHAN, S. 1998. A network architecture for heterogeneous mobile computing. *IEEE Pers. Comm. 5*, 5 (Oct.), 8–24.

BRINCH HANSEN, P. 1970. The nucleus of a multiprogramming system. *Comm. ACM 13*, 4 (Apr.), 238–241, 250.

BRUMITT, B., MEYERS, B., KRUMM, J., KERN, A., AND SHAFER, S. 2000. EasyLiving: Technologies for intelligent environments. In *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing*. Lecture Notes in Computer Science, vol. 1927. Springer-Verlag, Bristol, England, 12–29.

BULLARD, V., SMITH, K. T., AND DACONTA, M. C. 2001. *Essential XUL Programming*. John Wiley & Sons.

CARDELLI, L. 1999. Abstractions for mobile computations. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, J. Vitek and C. D. Jensen, Eds. Lecture Notes in Computer Science, vol. 1603. Springer-Verlag, 51–94.

CARRIERO, N. AND GELERNTER, D. 1986. The S/Net's Linda kernel. *ACM Trans. Comput. Syst. 4*, 2 (May), 110–129.

CHANKHUNTHOD, A., DANZIG, P. B., NEEDAELS, C., SCHWARTZ, M. F., AND WORRELL, K. J. 1996. A hierarchical Internet object cache. In *Proceedings of the 1996 USENIX Annual Technical Conference*. San Diego, CA. 153–163.

CHOU, P., ORTEGA, R., HINES, K., PARTRIDGE, K., AND BORRIELLO, G. 1999. ipChinook: An integrated IP-based design framework for distributed embedded systems. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*. New Orleans, LO. 44–49.

CZERWINSKI, S. E., ZHAO, B. Y., HODES, T. D., JOSEPH, A. D., AND KATZ, R. H. 1999. An architecture for a secure service discovery service. In *Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*. Seattle, WA. 24–35.

DAVIES, N., FRIDAY, A., WADE, S. P., AND BLAIR, G. S. 1998. L^2imbo: A distributed systems platform for mobile computing. *Mobile Networks and Applications 3*, 2 (Aug.), 143–156.

DERTOUZOS, M. L. 1999. The future of computing. *Sci. Amer. 281*, 2 (Aug.), 52–55.

DOUGLIS, F. AND OUSTERHOUT, J. 1991. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice and Experience 21*, 8 (Aug.), 757–785.

ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. 1995. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain Resort, CO. 251–266.

ESLER, M., HIGHTOWER, J., ANDERSON, T., AND BORRIELLO, G. 1999. Next century challenges: Data-centric networking for invisible computing. In *Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*. Seattle, WA. 256–262.

FERGUSON, P., LEMAN, G., PERINI, P., RENNER, S., AND SESHAGIRI, G. 1999. Software process improvement works! Tech. Rep. CMU/SEI-99-TR-027 (Nov.). Carnegie Mellon University, Software Engineering Institute.

FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. 1996. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*. Seattle, WA. 137–151.

FOSTER, I. AND KESSELMAN, C. 1997. Globus: A metacomputing infrastructure toolkit. *Inter. J. Supercomput. Appl. High Perf. Comput. 11*, 2, 115–128.

FOWLER, R. J. 1985. Decentralized object finding using forwarding addresses. Ph.D. thesis, University of Washington.

FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. 1997. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. Saint-Malo, France, 78–91.

FREEMAN, E., HUPFER, S., AND ARNOLD, K. 1999. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley.

GARLAN, D., SIEWIOREK, D. P., SMAILAGIC, A., AND STEENKISTE, P. 2002. Project Aura: Toward distraction-free pervasive computing. *IEEE Perv. Comput. Mag. 1*, 2 (April–June), 22–31.

GAST, M. 2002. *802.11 Wireless Networks: The Definitive Guide*. O'Reilly.

GONG, L. 1999. *Inside Java Platform Security—Architecture, API Design, and Implementation*. Addison-Wesley.

GRAY, C. G. AND CHERITON, D. R. 1989. Leases: An efficient fault-tolerant mechanism for file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*. Litchfield Park, AZ. 202–210.

GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. 1996. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. Montreal, Canada, 173–182.

GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. 2000. Scalable, distributed data structures for Internet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*. San Diego, CA. 319–332.

GRIMM, R. 2000. System support for pervasive applications. Ph.D. thesis, University of Washington.

GRIMM, R. AND BERSHAD, B. N. 2001. Separating access control policy, enforcement and functionality in extensible systems. *ACM Trans. Comput. Syst. 19*, 1 (Feb.).

GRIMM, R., DAVIS, J., LEMAR, E., MACBETH, A., SWANSON, S., GRIBBLE, S., ANDERSON, T., BERSHAD, B., BORRIELLO, G., AND WETHERALL, D. 2001. Programming for pervasive computing environments. Tech. Rep. UW-CSE-01-06-01 (June). University of Washington.

GUTTMAN, E., PERKINS, C., VEIZADES, J., AND DAY, M. 1999. Service location protocol, version 2. RFC 2608, Internet (June). Engineering Task Force.

GUZDIAL, M. AND ROSE, K., Eds. 2002. *Squeak: Open Personal Computing and Multimedia*. Prentice Hall.

HEIDEMANN, J. S. AND POPEK, G. J. 1994. File-system development with stackable layers. *ACM Trans. Comput. Syst. 12*, 1 (Feb.), 58–89.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. 2000. System architecture directions for networked sensors. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA. 93–104.

IOANNIDIS, J. AND MAGUIRE, JR., G. Q. 1993. The design and implementation of a mobile internetworking architecture. In *Proceedings of 1993 Winter USENIX Conference*. San Diego, CA. 491–502.

JOHANSON, B., FOX, A., AND WINOGRAD, T. 2002. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Perv. Comput. Mag. 1*, 2 (April–June), 67–74.

JOHNSON, K. L., CARR, J. F., DAY, M. S., AND KAASHOEK, M. F. 2000. The measured performance of content distribution networks. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*. Lisbon, Portugal. `http://www.terena.nl/conf/wcw/Proceedings/S4/S4-1.pdf`.

JONES, M. B. 1993. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. Asheville, NC. 80–93.

JOSEPH, A. D., DELESPINASSE, A. F., TAUBER, J. A., GIFFORD, D. K., AND KAASHOEK, M. F. 1995. Rover: A toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain Resort, CO. 156–171.

JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.), 109–133.

KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICEÑO, H., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. 1997. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. Saint-Malo, France, 52–65.

KISTLER, J. J. AND SATYANARAYANAN, M. 1992. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb.), 3–25.

KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. 2000. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA. 190–201.

KUMAR, P. AND SATYANARAYANAN, M. 1995. Flexible and safe resolution of file conflicts. In *Proceedings of the 1995 USENIX Annual Technical Conference*. New Orleans, LO. 95–106.

LANGE, D. B. AND OSHIMA, M. 1998. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley.

LE HORS, A., LE HÉGARET, P., WOOD, L., NICOL, G., ROBIE, J., CHAMPION, M., AND BYRNE, S. 2000. Document object model (DOM) level 2 core specification. W3C recommendation, World Wide Web Consortium (Nov.). Cambridge, MA.

LEACH, P. J. AND SALZ, R. 1998. UUIDs and GUIDs. Internet Draft draft-leach-uuids-guids-01.txt (Feb.). Internet Engineering Task Force.

LEVY, E. AND SILBERSCHATZ, A. 1990. Distributed file systems: Concepts and examples. *ACM Comput. Surv. 22*, 4 (Dec.), 321–374.

LEWIS, M. AND GRIMSHAW, A. 1996. The core Legion object model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. Syracuse, NY. 551–561.

LIANG, S. AND BRACHA, G. 1998. Dynamic class loading in the Java virtual machine. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications '98*. Vancouver, Canada, 36–44.

LIEFKE, H. AND SUCIU, D. 2000. XMill: An efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. Dallas, TX. 153–164.

LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification*, Second ed. Addison-Wesley.

MARTIN, B. AND JANO, B. 1999. WAP binary XML content format. W3C note, World Wide Web Consortium (June). Cambridge, MA.

MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. 1996. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley.

MICROSOFT CORPORATION. 1999. *Microsoft Office 2000 Resource Kit*. Microsoft Press.

MICROSOFT CORPORATION. 2000. Understanding universal plug and play. White paper (June). Microsoft Corporation, Redmond, WA. Available at `http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc`.

MILLSTEIN, T. 2004. Practical predicate dispatch. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Vancouver, BC. 345–364.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. MIT Press.

MILOJIČIĆ, D., DOUGLIS, F., AND WHEELER, R., EDS. 1999. *Mobility—Processes, Computers, and Agents*. ACM Press. Addison-Wesley.

MUMMERT, L. B., EBLING, M. R., AND SATYANARAYANAN, M. 1995. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain Resort, CO. 143–155.

MURPHY, A. L., PICCO, G. P., AND ROMAN, G.-C. 2001. Lime: A middleware for physical and logical mobility. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*. Phoenix, AZ. 524–533.

MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. 2001. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*. Banff, Canada, 174–187.

NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. 1997. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. Saint-Malo, France, 276–287.

NYE, A., Ed. 1995. *X Protocol Reference Manual*, 4th ed. O'Reilly.

OKI, B., PFLUEGL, M., SIEGEL, A., AND SKEEN, D. 1993. The Information Bus—an architecture for extensible distributed systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. Ashville, NC. 58–68.

OLSON, M. A., BOSTIC, K., AND SELTZER, M. 1999. Berkeley DB. In *Proceedings of the FREENIX Track, 1999 USENIX Annual Technical Conference*. Monterey, CA. 183–192.

OUSTERHOUT, J. 1996. Why threads are a bad idea (for most purposes). `http://home.pacbell.net/ouster/threads.ppt`. Invited Talk presented at the *1996 USENIX Annual Technical Conference*. San Diego, CA.

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Annual Technical Conference*. Monterey, CA. 199–212.

PARDYAK, P. AND BERSHAD, B. N. 1996. Dynamic binding for an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*. Seattle, WA. 201–212.

PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. 1997. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. Saint-Malo, France, 288–301.

PETZOLD, C. 1998. *Programming Windows*, 5th ed. Microsoft Press.

PLAINFOSSÉ, D. AND SHAPIRO, M. 1995. A survey of distributed garbage collection techniques. In *International Workshop on Memory Management*. Lecture Notes in Computer Science, vol. 986. Springer-Verlag, Kinross, Scotland, 211–249.

POWELL, M. L. AND MILLER, B. P. 1983. Process migration in DEMOS/MP. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*. Bretton Woods, NH. 110–119.

PRECHELT, L. 2000. An empirical comparison of seven programming languages. *IEEE Comput. 33*, 10 (Oct.), 23–29.

RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. 2003. Pond: The OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies.* San Francisco, CA. 1–14.

RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. 1998. Virtual network computing. *IEEE Inter. Comput. 2*, 1 (January/February), 33–38.

SALTZER, J. H., REED, D. P., AND CLARK, D. D. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems 2*, 4 (Nov.), 277–288.

SATOH, I. 2000. MobileSpaces: A framework for building adaptive distributed applications using a hierarchical mobile agent system. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*. Taipei, Taiwan, 161–168.

SATYANARAYANAN, M. 2001. Pervasive computing: Vision and challenges. *IEEE Pers. Comm. 8*, 4 (Aug.), 10–17.

STEENSGAARD, B. AND JUL, E. 1995. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain Resort, CO. 68–77.

STEVENS, W. R. 1994. *TCP/IP Illustrated*. Vol. 1. Addison-Wesley.

SUN MICROSYSTEMS. 2002. Java remote method invocation specification. Revision 1.8, Sun Microsystems, Palo Alto, CA.

TAMCHES, A. AND MILLER, B. P. 1999. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*. New Orleans, LO. 117–130.

TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain Resort, CO. 172–183.

TEWARI, R., DAHLIN, M., VIN, H. M., AND KAY, J. S. 1999. Design considerations for distributed caching on the Internet. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*. Austin, TX. 273–284.

THAI, T. AND LAM, H. 2002. *.NET Framework Essentials*, 2nd Ed. O'Reilly.

THOMPSON, H. S., BEECH, D., MALONEY, M., AND MENDELSOHN, N. 2001. XML schema part 1: Structures. W3C recommendation, World Wide Web Consortium (May). Cambridge, Massachusetts.

THORN, T. 1997. Programming languages for mobile code. *ACM Comput. Surv. 29*, 3 (Sept.), 213–239.

TULLMANN, P. AND LEPREAU, J. 1998. Nested Java processes: OS structure for mobile code. In *Proceedings of the 8th ACM SIGOPS European Workshop*. Sintra, Portugal, 111–117.

TULLOCH, M. 2001. *Windows 2000 Administration in a Nutshell*. O'Reilly.

VAN STEEN, M., HOMBURG, P., AND TANENBAUM, A. S. 1999. Globe: A wide-area distributed system. *IEEE Concurr. 7*, 1, 70–78.

WALRATH, K. AND CAMPIONE, M. 1999. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley.

WEISER, M. 1991. The computer for the twenty-first century. *Sci. Amer. 265*, 3 (Sept.), 94–104.

WELSH, M., CULLER, D., AND BREWER, E. 2001. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*. Banff, Canada, 230–243.

WYCKOFF, P., MCLAUGHRY, S. W., LEHMAN, T. J., AND FORD, D. A. 1998. T Spaces. *IBM Syst. J. 37*, 3, 454–474.

Received February 2002; revised June 2003, May 2004; accepted September 2004