

# Migration for Pervasive Applications

Robert Grimm, Janet Davis, Eric Lemar, and Brian Bershad  
*University of Washington*  
{rgrimm, jlnd, elemar, bershad}@cs.washington.edu

## Abstract

This paper explores migration, the ability to move an application between devices, in the context of pervasive, or ubiquitous, computing. Pervasive computing, with its vision of information access anytime and anywhere, introduces a new class of applications that seamlessly follow people as they move through the physical world. As a result, migration becomes an important building block for the development and deployment of pervasive applications. However, the main challenge with migration is to strike a reasonable balance between the complexity of implementing migration and the actual utility delivered to applications. In this paper, we introduce a migration service for pervasive applications that marks a new, practical design point in the design space. We explain the trade-offs made in its design, detail its implementation, and reflect on our own and others' experiences with using it.

## 1 Introduction

In this paper, we present a migration service specifically designed for pervasive computing environments. The key idea behind pervasive, or ubiquitous, computing [26] is to deploy a wide variety of computing devices throughout our living and working spaces. These devices then coordinate with each other and network services [6], with the goal of seamlessly assisting people in completing their tasks. In such an environment, migration becomes an essential system service, allowing applications to follow people as they move about during the course of their day and remain continuously available.

A considerable number of projects have explored how to provide migration [19]—the ability

to move an application and its state from one device to another. Yet, despite a large body of considerably different designs, which include process migration as provided by Sprite [7], object mobility as provided by Emerald [15, 23], and mobile agents as provided by Aglets [16], migration has not been widely adopted as a system service. This lack of widespread acceptance stems from the fact that traditional uses of migration, such as exploiting the computing power of idle workstations, are not sufficiently compelling and, as an application domain, not specific enough.

As a result, these projects had limited success in defining reasonable requirements for migration and in finding the right trade-off between the complexity of implementing migration and the actual utility delivered to applications. At one extreme, Douglass and Ousterhout conclude about Sprite's support for transparent process migration [7]:

From the outset we expected migration to be difficult to build and maintain. Even so, we were surprised at the complexity of the interactions between process migration and the rest of the kernel.

At the other extreme, most mobile agent systems are considerably less complex, but provide little functionality beyond serializing an agent's objects on one device and then deserializing them on another.

In contrast, pervasive computing as an application domain provides a compelling use for migration: applications that follow people as they move through the physical world. This focus lets us define clear-cut requirements for an appropriate migration service. First, migration needs to be visible to applications rather than being transparent, so that applications know their execution context and can

adapt to it. Second, migration needs to integrate persistent storage, so that applications can continuously provide access to people's information. Third, migration must be easy to control, so that migration logic can be effectively factored from the rest of application functionality and devices protected against malicious applications roaming around the network. Our focus on applications that follow people as they move about also helps set performance goals for migration: It needs to perform well enough to match people's movements in the physical world.

Of course, developers of pervasive applications could implement their own, application-specific migration mechanisms. That way, they only need to implement as much functionality as strictly needed by their applications. They might even be able to avoid moving execution state altogether and rather just communicate necessary data. However, this approach also places a considerable burden on application developers. They need to correctly implement their application-specific migration mechanisms and evolve them as their applications evolve. Furthermore, in the absence of full migration, they need to deploy their applications throughout an environment with a plethora of (embedded) devices. In other words, without migration as a system service, the application developers' task becomes considerably harder.

We thus introduce a practical migration service for pervasive applications. Our migration service is part of a larger system architecture for pervasive applications, called *one.world* [12]. It builds on our architecture's environment service, which provides a nestable container for both computations and storage, and moves or copies environments between devices. By integrating migration with *one.world*'s environments, our service can easily meet all three requirements. Environments provide a clean model for what state not to migrate (thus making migration visible), integrate persistent storage, and, through nesting, provide a powerful mechanism for controlling which environments can migrate when and where. Our own and others' experiences show that our migration service is powerful, easy-to-use, and generally useful for building system services, utilities, and applications alike.

The rest of this paper is structured as follows.

We start by presenting an example application in Section 2. We then introduce *one.world* and its services, including migration, in Section 3. We follow with a detailed discussion of the design and implementation of our migration service in Sections 4 and 5, respectively. We then reflect on our own and others' experiences with using migration in Section 6 and present performance results in Section 7. We compare *one.world*'s migration with representative previous designs in Section 8. Finally, we conclude in Section 9.

## 2 An Example Application

To illustrate migration for pervasive applications, we now introduce the digital laboratory. This application addresses a real need of real people—performing reproducible biology experiments. Furthermore, as discussed in Section 6.3, the digital laboratory has been implemented on top of *one.world* by the University of Washington's Labscape project and deployed at the Cell System Initiative [3]. As a result, it provides an apt example for using our migration service.

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' movements, and touchscreens to display experimental data close to the researchers. 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. Clearly, migration is directly useful for im-

plementing the digital laboratory application. At the same time, there is no need to migrate the entire digital laboratory application. Rather, only a small component to capture and display experimental data needs to be migrated. Eventually, all data is forwarded to a centralized repository, making it possible to, for example, mine the data of several experiments.

The three requirements for migration show up in the digital laboratory as follows. First, as the capture and display component follows a researcher, it needs to connect and re-connect to close-by instruments (which typically change as the location changes). Consequently, migration should notify the component of location changes and not maintain connections from the previous location. Second, the capture and display component needs to be able to function without access to the repository, for example, when the researcher is commuting home, and reliably forward updates upon reconnection. As a result, the capture and display component must have access to persistent storage that is moved with itself. Finally, to implement the digital laboratory, migration needs to allow all active capture and display components to roam around the laboratory. It should also integrate individual researchers’ tablet computers and laptops. However, to provide a secure environment, it must not allow any other, potentially malicious applications to utilize the laboratory’s computing resources.

### 3 Overview of *one.world*

*one.world* provides system support for pervasive applications and is centered around meeting three major requirements. First, as people move throughout the physical world—either carrying their own portable devices or switching between devices—an application’s location and execution context changes all the time. As a result, system support needs to *embrace contextual change* and not hide it from applications. Second, users expect that their devices and applications just plug together. System support thus needs to *encourage ad hoc composition* and not assume a static computing environment with a limited number of interactions. Third, as users collaborate, they need to easily share informa-

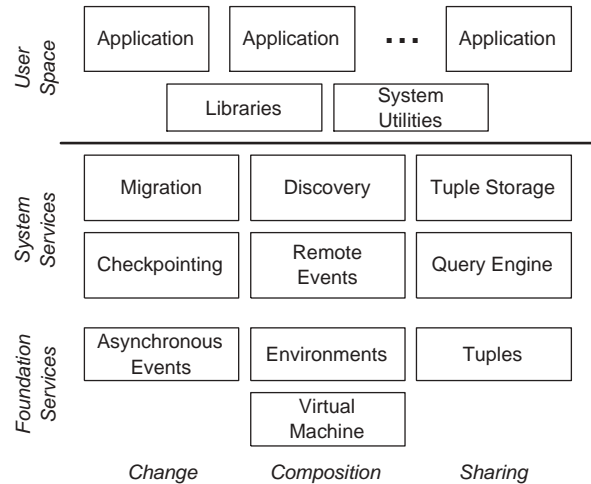


Figure 1: Overview of *one.world*’s architecture. Foundation and system services are part of the kernel, while libraries, system utilities, and applications run in user space.

tion. As a result, system support needs to *facilitate sharing* between applications and devices.

Our architecture is illustrated in Figure 1. It employs a classic user/kernel split, with foundation and system services being provided by the kernel, and libraries, system utilities, and applications running in user space. *one.world*’s foundation services directly address the individual requirements and provide the basis for our architecture’s system services. The system services, in turn, serve as common building blocks for pervasive applications.

The four foundation services are a virtual machine, tuples, asynchronous events, and environments. First, all code in *one.world* runs within a virtual machine, such as the Java virtual machine [18] or Microsoft’s common language runtime [24]. Because of the inherent heterogeneity of pervasive computing environments, developers cannot possibly predict all devices their applications will run on, and the virtual machine ensures that applications and devices are composable. Second, all application data in *one.world* is represented as tuples. Tuples are self-describing in that an application can dynamically inspect a tuple’s structure. They also define a type system common to all applications, thus simplifying the sharing of data. Third, all communications in *one.world*, whether local or remote,

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

Table 1: Application needs and corresponding system services.

are expressed through asynchronous events. They provide the means for explicitly notifying applications of changes in their runtime context.

Finally, environments are the main structuring mechanism for *one.world* and serve as nestable containers for computations and persistent data alike. They can be thought of as a combination of file system directories and nested processes [4, 8, 25] in other operating systems. As containers for local computations, environments are isolated from each other. Isolation is enforced by copying all data across environments; the only exception are references to event handlers, which can be directly exchanged so that applications can utilize each others' services. Environments also are the containers for persistently stored tuples. As a result, applications and their data can be directly grouped with each other. Furthermore, environments can be nested within each other, making it easy to compose applications. An outer environment has complete control over all nested environments, including the ability to interpose on inner environments' communications with *one.world*'s kernel and the outside world.

*one.world*'s system services build on the foundation services and serve as common building blocks for pervasive applications. Specific application needs and the corresponding system services are summarized in Table 1. Out of these services, discovery and migration are the most relevant for this paper.

Discovery locates event handlers and routes events to them. It leverages *one.world*'s uniform data model, in which all data, including events and queries, are tuples, to support a rich set of options with only three simple operations. Options include early and late binding [2] as well as any-

cast and multicast. More importantly, because pervasive applications require discovery to be continuously available—after all, without discovery, applications cannot adapt to a new or changing execution context—discovery is self-managing and relies on an elected discovery server.

Migration moves or copies an environment and all its contents, that is, the active computations, stored tuples, and nested environments. Unlike traditional process migration, *one.world*'s migration is not transparent, and migrated application state is limited to the environments being migrated. References to resources outside the environment tree are automatically nulled out during migration. After migration has completed, the migrated environments are explicitly notified of the migration and can then reconnect to outside resources as necessary, typically by using discovery.

The implementation of *one.world* is written mostly in Java, though it relies on the Berkeley DB [20] for atomic tuple storage. Our source tree comprises about 109,000 lines of code or 40,000 statements, representing approximately 6 man years of development. A Java archive file with the binaries for *one.world* itself is 514 KB. The platform-native libraries require another 528 KB on Windows and 805 KB on Linux systems. Our architecture has been released as open source. It can be downloaded at <http://one.cs.washington.edu> and is currently at version 0.7.1.

## 4 Design

The design of *one.world*'s migration service is based on the assumption that pervasive computing environments often exhibit weaker connectivity than traditional local networks. We believe that disconnected operation may be a relatively frequent occurrence, and that connections, such as those using cell phones, may have high latency and low bandwidth. As a result, *one.world*'s migration is not only application-visible (as opposed to transparent) but also eager. It moves or copies an environment and its contents in one atomic operation. Furthermore, it does not maintain any forwarding addresses [9] or residual dependencies [21] across

the network. References from within the migrating environment tree to outside resources are nulled out during migration. After migration has completed, the migrated environments are notified that they have been moved or copied, so that they can replace nulled out resources. References from the outside into the migrating environment tree are invalidated during a move but not a copy operation. When sending an event to such a reference, the sender is notified that the resource has been moved.

With the basic structuring of migration in place, we now turn to the important issues of (1) what state to migrate, (2) what state not to migrate, and (3) how to control migration.

#### 4.1 Included State

As already mentioned, migration moves or copies an environment and all its contents. This obviously includes the environment itself and all nested environments. It also includes all tuples persistently stored in the migrating environment tree, thus fulfilling the requirement that migration integrate persistent storage. Finally, this includes the state of all computations in the environment tree.

To move or copy this state, migration captures a checkpoint—a bytestring representing the computations’ state—on the sending side and restores the checkpoint on the receiving side. The checkpointing mechanism 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 checkpointing mechanism captures the state of the computations’ objects.

Since all communications in *one.world* are through asynchronous events, the checkpointing mechanism also captures the state of event processing by serializing  $\langle event\ handler, event \rangle$  invocations. Comparable to bus stops in Emerald [23], which define application states that are safe to migrate, execution state can only be captured for pending  $\langle event\ handler, event \rangle$  invocations. Invocations that are currently being executed need to run to completion, because capturing them requires access to the virtual machine’s execution stack.

However, many virtual machines, like the Java virtual machine [18] but unlike the Squeak virtual machine [14], do not explicitly expose their execution stacks. Capturing currently executing  $\langle event\ handler, event \rangle$  invocations would thus require modifications to the underlying virtual machine.

#### 4.2 Excluded State

Moved or copied state is limited to the environment being migrated, and, as already stated, no forwarding addresses or residual dependencies are maintained for migrated state, thus fulfilling the requirement that migration be visible to applications. Since environments are isolated from each other, only references to event handlers can be exchanged between environments; all other data is copied. As a result, the migrated state can be limited simply by testing event handlers while creating the checkpoint. If an event handler is part of the migrating environment 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 what state to exclude from migration, and nulling out event handlers provides a simple contract for revoking such state.

#### 4.3 Controlling Migration

Environments provide control over migration, both on the sending and on the receiving side. One important feature of environments is that an outer environment has complete control over all nested environments, including the ability to interpose on their communications with *one.world*’s kernel. On the sending side, an outer environment can thus decide when and where to migrate a nested environment. It can also intercept a request to be migrated (that has been created by a nested environment) and either modify it or reject it. Similarly, on the receiving side, the future outer environments are notified 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 fulfills the requirement that migration be easy to control.

Environment nesting also enables an important

pattern for controlling migration. Under this pattern, the logic to decide when and where to migrate an application is factored into its own environment, and the actual application is nested within that environment. As a result, the 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: It leverages the environment nesting to provide the ability to move all of a user's applications between devices.

#### 4.4 Sources of Complexity

As discussed above, *one.world*'s migration leverages our architecture's foundation services as much as possible to avoid complexity and to provide a clean and useful model for migration. In particular, it relies on environments to delineate what state to migrate and what state not to migrate. It also relies on environments for controlling migration. Furthermore, it relies on asynchronous events to make the execution state explicit. Finally, it relies on the virtual machine to provide a uniform execution environment across different devices and hardware architectures.

At the same time, our migration service introduces a new source of complexity when compared to transparent migration services, because applications now need to explicitly reconnect to outside resources after migration. As a result, discovery becomes a central service for pervasive applications, and *one.world*'s discovery service has been carefully designed to expose an easy-to-program interface and to be self-managing. In *one.world*'s uniform data model, even events and queries are represented as tuples. The discovery service leverages this uniformity to support a rich set of options, including early and late binding, anycast, and multicast, with only three simple operations that (1) export an event handler to discovery, (2) look up an event handler, and (3) send an event. Furthermore, it relies on a discovery server that is automatically elected from the devices on a local network, thus removing the need for manual configuration and administration.

---

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

---

Figure 2: Example code for moving an environment. This code snippet moves a user's environment `user.env` to the device named `location` and the new parent environment "Users". The operation forwards the newly created event to *one.world*'s kernel. The code snippet is taken from Emcee's source code.

## 5 Implementation

Migration in *one.world* goes through three stages. In the first stage, an application decides where to migrate to and then initiates the migration. In the second stage, *one.world*'s kernel performs the actual move or copy operation. In the third stage, the application reconnects to outside resources by restoring nulled out event handlers. Since the first and third stages are application-specific, this section focuses on the inner workings of the second stage.

To migrate an environment, the application initiating the migration sends a migration request to *one.world*'s kernel (as illustrated in Figure 2). The request specifies the environment to be migrated, the receiving device and new parent environment on that device, and whether to move or copy the environment. Environments are named by either a globally unique identifier [17] (GUID) or a human-readable path name. Devices are named by their IP address or DNS name. When *one.world*'s kernel receives such a request, it verifies that the requesting environment is an ancestor of the environment to be migrated and then performs the move or copy operation.

At the core of the implementation of *one.world*'s migration service lies the protocol that communicates the migrating environment and its contents from the sending to the receiving device. The protocol is implemented using our architecture's remote events. It is organized into several rounds,

Sender	Receiver
1. Quiesce environments and create checkpoint.	
2. Send request.	
4. Send environment meta-data.	3. Submit request to future ancestral environments and send confirmation.
6. Send stored tuple.	5. Create environment data structures and send confirmation.
8. Send checkpoint.	7. Tentatively store tuple and send confirmation.
	9. Restore checkpoint, commit stored tuples, enqueue move/copy notification, activate environments, and send confirmation.
10. On move, destroy migrated environments; on copy or when requesting environment has not been moved, notify requesting environment of completed migration.	

Figure 3: Summary of *one.world*'s migration protocol. Time flows downwards from the top of the figure. Note that steps 6 and 7 are performed zero or more times, depending on the number of tuples stored in the migrating environment tree.

where each event issued by the sender is confirmed by the receiver. If an event is lost or an error occurs, the protocol and consequently the migration are aborted. The migration protocol is summarized in Figure 3; we use the step numbers in the text to refer back to this summary.

After receiving and verifying a migration request as described above, the sender prepares for the move or copy operation by quiescing all environments to be migrated (step 1). In particular, it lets all currently active event handler invocations run to completion and prevents new invocations from being executed. Once the environment tree is quiesced, it creates the checkpoint as described in Section 4, capturing the state of all application objects as well as the queues of pending *(event handler, event)* invocations. Note that, on copy operations, the copied environments are assigned freshly generated GUIDs to avoid duplicate identifiers, although the human-readable names remain the same. After the environment tree has been checkpointed, the sender is ready to start the migration protocol.

The sender first issues a request specifying the name of the migrating environment and the identity of the new parent (step 2). It then sends the meta-data for the migrating environment tree (step 4).

Next, it sends all stored tuples (step 6). Note that, to avoid the performance overhead of serialization, tuples are not deserialized and serialized when being passed between storage and the network. Rather, they are directly forwarded as bytestrings. Finally, the sender issues the checkpoint (step 8). Once the checkpoint has been confirmed, the migration protocol has successfully completed, and the sender can clean up its internal state (step 10). In particular, it destroys the migrated environment tree on a move operation, which invalidates references from the outside into the migrated environment tree. Furthermore, on a copy operation or if the requesting environment is outside the moved tree, the sender notifies the requesting environment of the successfully completed migration.

When receiving a migration request, the receiver submits the request to all future ancestral environments, which can either redirect the migrating environment tree or reject it (step 3). Note that, in the current implementation, requests are accepted by default. In contrast, a production system should reject migration requests by default and, for obvious security reasons, only accept properly authenticated environments. Once accepted, the receiver confirms the request to the sender. After receiving the meta-data for the migrating environment

tree, the receiver creates the appropriate internal data structures and issues a confirmation (step 5). Next, after receiving a tuple, the receiver tentatively stores that tuple and confirms that it has received the tuple (step 7). Finally, after receiving the checkpoint, the receiver restores the checkpoint (step 9). Once the checkpoint has been restored, the migration has succeeded. The receiver commits all stored tuples, enqueues notifications of the completed migration, activates the migrated environments, and finally sends confirmation to the sender.

Our implementation does not currently handle program binaries, that is, the applications’ Java class data. Rather, Java classes are loaded from a device’s local classpath and not migrated. In contrast, *one.world*’s original design calls for storing class data as tuples in an application’s environment and then migrating the class data with the application. However, we believe that this design is too simple, as class data shared between applications is stored several times and may be repeatedly migrated. We thus suggest an improved design that still stores class data as tuples in individual environments but also backs class data by a shared cache, effectively storing each class only once. With the improved design, class data is only migrated if the necessary classes are not already in the cache.

The complexity of *one.world*’s migration service, as measured by its code size, is quite reasonable. The code for checkpointing and the migration protocol comprises about 1200 statements out of about 2500 statements for all environment operations. The code for migrating stored tuples comprises an additional 300 statements out of about 2000 statements for all tuple storage operations. Furthermore, there is some cost for making all core objects serializable, typically less than 10 statements per class. At the same time, *one.world*’s migration, just as for previous migration systems, does interact with several aspects of our architecture’s kernel, resulting in five tightly interdependent classes at the core of *one.world*.

## 6 Experiences With Migration

In this section, we discuss how we and others have used migration in building pervasive applications

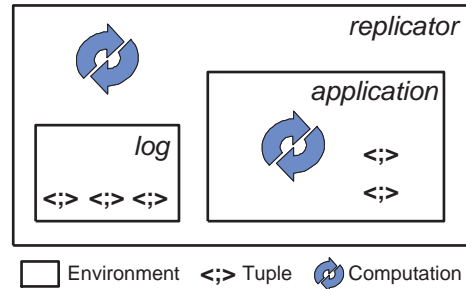


Figure 4: Structure of our replication service. The “replicator” environment intercepts all storage operations issued by the “application” environment. In disconnected mode, updates are logged in the “log” environment.

and supporting infrastructure. We present three examples that use migration in different ways. First, our replication layer uses migration as an internal building block [13]. Second, our application manager Emcee controls how other applications are migrated. Third, in the Labscape digital laboratory, the capture and display component moves itself [3]. Taken together, these examples illustrate the flexibility and power afforded by our migration service. More importantly, they show that our migration service provides a general building block for system services, utilities, and applications alike.

### 6.1 Replication Service

Our replication service [13] makes stored tuples accessible across several devices, even if they are disconnected. It is patterned after Gray et al.’s two-tier replication model [10], and relies on a master to own all data. The replication service runs in user space and, as illustrated in Figure 4, exploits the environment nesting 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 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, migra-

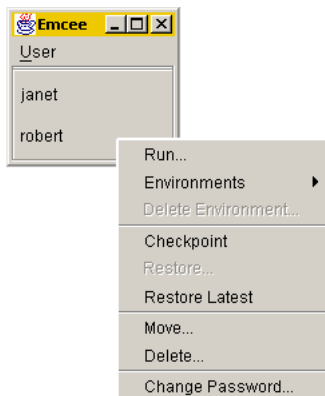


Figure 5: A screenshot of Emcee’s user interface. The main window lists the users whose applications run on a device. A popup menu for each user, shown for the user named “robert”, is used to perform most operations, such as running a new application or checkpointing a user’s applications. The user menu supports the creation of new users and the fetching of a user’s applications from another device.

tion 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.

## 6.2 Emcee

Emcee, whose user interface is shown in Figure 5, is our Finder-like user manager. It includes support for creating new users, running applications for a user, and checkpointing all of a user’s applications. It also provides the ability to move or copy applications between users. However, Emcee’s most interesting feature is the ability to move all of a user’s applications between devices. Applications can either be pushed from the current device, or they can

be pulled from another device.

Emcee’s implementation structures the environment hierarchy according to the pattern `/Users- /<user>/<application>`. Emcee runs in the `/Users` environment and uses a child environment for each user and a grandchild for each application. Pushing a user’s applications is straightforward as it directly utilizes our migration service. However, fetching a user’s applications from a remote device requires a two-round protocol to properly authenticate the user to the remote instance of Emcee. After the user has been successfully authenticated, the remote Emcee migrates the user’s environment tree to the requesting device. If the user’s location is not specified the initial remote event for the fetcher protocol is routed through late binding discovery. Otherwise, it is sent directly to the remote device.

Emcee illustrates the power of combining migration with environment nesting for control. Emcee controls the location of a user’s applications simply by nesting the applications in its environment. Applications do not need their own migration logic. Furthermore, they 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.

## 6.3 Labscape

The Labscape project, in cooperation with biology researchers, has implemented the digital laboratory application outlined in Section 2 on top of *one.world* [3]. The Labscape application is currently in use at the University of Washington’s Cell Systems Initiative and will soon be deployed at Immunex Corporation.

As already discussed, the Labscape application is structured into a capture and display component, called a *guide* and illustrated in Figure 6, and a backend repository to collect all experimental data. It also relies on a so-called *proximity service* to track researchers’ movements through the laboratory and to correlate instrument readings with the researchers performing them. Under this design, location sensors and instruments forward their readings to the proximity service. The proximity service, in turn, forwards instrument readings to the

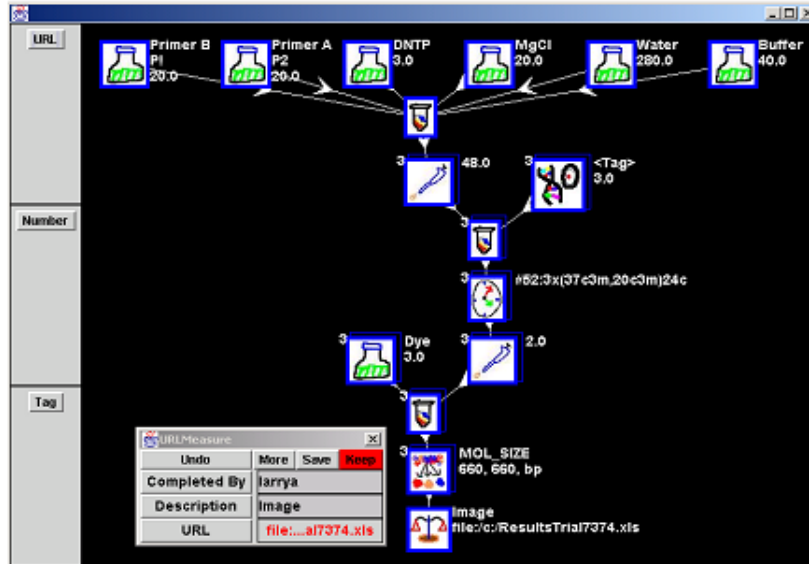


Figure 6: A screenshot of Labscape’s user interface, which represents a single biology experiment. The visual representation is organized as a flowgraph, with icons representing individual steps and lines representing ordering constraints. Initially, the flowgraph serves as a plan for the experiment to be performed. With each step, the corresponding icon is annotated with relevant information, and, over time, the flowgraph becomes a record of the experiment.

appropriate guide. It also notifies a guide when the researcher has changed location. In response, the guide follows the researcher, simply by initiating a move to the next closest touchscreen. Migration thus becomes just another building block in the overall application.

The Labscape application clearly demonstrates the utility of our migration service for implementing real pervasive applications. Furthermore, the history of the Labscape project underlines the importance of providing appropriate system support, including migration, for pervasive applications. Before porting the digital laboratory to *one.world*, the Labscape team implemented their application directly in Java, using TCP sockets for communications and their own, application-specific migration layer to move the guide [3].

In comparison to the *one.world* version, the Java version of Labscape is slow and fragile. In particular, every major modification of the guide required corresponding changes in the migration layer. Yet, despite the application-specific migration layer, moving the guide in the Java version is five to ten times slower than in the *one.world* ver-

sion. Furthermore, the Java version of the Labscape application 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. At the same time, system support can help with graceful failure recovery. For instance, after a failure in 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 the piecemeal restarting of components and thus is considerably more resilient in the face of buggy application code.

## 7 Performance Evaluation

To determine whether migration performs well enough to match people’s movements in the physical world, we measured real applications as well as a set of micro-benchmarks. In summary, these experiments show that *one.world*’s migration ser-

vice performs acceptably. However, they also suggest that the migration protocol could be rewritten to better exploit available bandwidth.

All measurements reported in this section, with exception of the Labscape measurements, were taken on 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 end-to-end migration performance, we measured Chat—a text and audio messaging application we have developed—as it migrates under the control of Emcee. When listening to music, playback is interrupted for 3.7 seconds. During this time, Chat migrates between two devices, re-initializes audio, and re-registers with discovery (all of Chat's communications are through late binding discovery). In [3], the Labscape team reports migration times between 2.5 seconds for migrating a guide with no experimental data and 7.1 seconds for migrating a guide with 64 samples, representing a large experiment. These experiments show that migration is fast enough when compared to a person moving through the physical world.

To further quantify the scalability of migration, we also 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 should optimistically stream tuples and thus overlap the individual steps instead of moving one tuple per protocol round.

## 8 Related Work

An exhaustive review of previous work on migration is beyond the scope of this paper; instead, we refer the reader to Milojević et al.'s excellent *Mobility* [19]. In this section, we focus on three representative, “best-of-breed” systems to highlight the most important points in the design space. The three systems are transparent process migration as provided by Sprite [7], object mobility as provided by Emerald [15, 23], and mobile agents as provided by Aglets [16]. Table 2 provides a comparison of these systems with *one.world*'s migration service.

Probably the most important differentiating factor between Sprite and Emerald on one side and Aglets and *one.world* on the other side is whether migration is transparent. Both Sprite and Emerald target local networks and, based on the assumption that such networks are well-maintained and experience few failures, seek to provide transparent migration. They thus use forwarding addresses [9] and residual dependencies [21] to hide the effects of location changes (though, location itself is visible in Emerald), which results in considerable system complexity. Sprite's transparent integration with distributed file storage and Emerald's transparent support for multiple instruction sets represent additional sources of complexity.

In contrast, migration in Aglets and *one.world* is not transparent. They migrate only an agent or environment, respectively, and can thus avoid most of the complexity of the other systems. However, Aglets' functionality also is rather limited, as it does not migrate execution state (though, the Telescript mobile agent system [27] does migrate execution state) and does not integrate storage beyond the ability to save agents. *one.world* differs from Aglets in that it migrates both execution state and persistent data. Furthermore, it differs from

	<b>Sprite</b>	<b>Emerald</b>	<b>Aglets</b>	<b><i>one.world</i></b>
Target network	Local network	Local network	Internet	Local and wide area networks
Execution environment	Single instruction set	Multiple instructions sets	Virtual machine	Virtual machine
Unit of migration	Process	Object (including all “attached” objects)	Agent	Environment
Control over migration	Parent process	Any object with reference	Any object with reference, security policy on receiving side	Environment and its parents, both on sending and receiving side
Transparency	Yes, every machine appears just like the “home” machine	Yes, but location is visible	No, only agent with internal objects is migrated	No, only environment with its contents is migrated
Execution model	Process-based	Thread-based	Thread-based	Event-based
Migrated execution state	Stack	Stack	None	Event queue
Integration with storage	Distributed file system	None	None, but ability to save agents	Local tuple storage

Table 2: Comparison of representative migration services with *one.world*’s migration service.

all three systems in that the environment hierarchy provides a well-defined and clean model for controlling which environments to move away from a device and which migrating environments to accept on a device.

In [22], Snoeren et al. introduce a different form of migration, which moves the end-point of a TCP connection to a different device (without tearing down the connection). TCP end-point migration is orthogonal and complimentary to the migration services discussed above. At the same time, in our experience with *one.world*, pervasive applications rarely use point-to-point communications. Instead, they typically communicate through late binding discovery and may additionally use multicast to address several components at the same time. As a result, we believe that TCP end-point migration is not as useful for pervasive applications as the environment migration provided by *one.world*.

## 9 Conclusions

In this paper, we have introduced a practical migration service for pervasive applications. Our migration service builds on *one.world*’s environment service and moves or copies environments be-

tween devices. In contrast to many previous migration systems, our migration service is application-visible and only moves or copies an environment and its contents. Furthermore, our migration service integrates with persistent storage as it migrates stored tuples in addition to an application’s execution state. Finally, through environment nesting, our migration service provides a clean and powerful mechanism for controlling which environments can migrate when and where. Our own and others experiences with *one.world*’s migration service show that it is indeed a viable and flexible building block for pervasive applications. Furthermore, its performance is good enough to match people’s movements in the physical world.

One issue we have not yet addressed is how to build applications that gracefully scale over the considerable range of hardware devices in pervasive computing environments. While scalability may impact an application’s feature set, we believe that scalability is most pressing for an application’s user interface. For instance, both Emcee and Labscape use Java’s Swing for implementing their user interfaces. Clearly, their interfaces will not scale down to cell phones or even PDAs, raising the question of how to build scalable user interfaces. One promising direction, suggested by UIML [1] and

the Mozilla project's XUL [5], is to replace such a programmatic specification of an application's interface with a declarative specification that is automatically rendered according to a device's input and output capabilities. Furthermore, as argued in [11], we believe that declarative specifications may be useful in implementing not only user interfaces but entire applications.

## Acknowledgments

We thank Tom Anderson and David Wetherall for providing us with valuable comments on earlier versions of this paper. In addition to the authors of this paper, Adam MacBeth, Steven Swanson, Daniel Cheah, Ben Hendrickson, Tom Anderson, Gaetano Borriello, Steven Gribble, and David Wetherall contributed to the development of *one.world*. This work was funded in part under DARPA contracts F30602-98-1-0205 and N66001-99-2-892401. Davis was partially supported by an NSF graduate fellowship. Davis, Grimm, and Lemar were partially supported by Intel Corporation internships. Grimm was also supported by IBM Corporation and Intel Foundation graduate fellowships.

## References

- [1] M. Abrams. User interface markup language (UIML). Draft specification, Harmonia, Inc., Blacksburg, Virginia, Jan. 2000. Available at <http://www.uiml.org/docs/uiml20>.
- [2] W. Adje-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island Resort, South Carolina, Dec. 1999.
- [3] L. Arnstein, R. Grimm, C.-Y. Hung, J. H. Kang, A. LaMarca, S. B. Sigurdsson, J. Su, and G. Borriello. Systems support for ubiquitous computing: A case study of two implementations of Labscape. In *Proceedings of the 2002 International Conference on Pervasive Computing*, Zurich, Switzerland, Aug. 2002.
- [4] P. Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 250, Apr. 1970.
- [5] V. Bullard, K. T. Smith, and M. C. Daconta. *Essential XUL Programming*. John Wiley & Sons, July 2001.
- [6] M. L. Dertouzos. The future of computing. *Scientific American*, 281(2):52–55, Aug. 1999.
- [7] F. Douglass and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice and Experience*, 21(8):757–785, Aug. 1991.
- [8] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 137–151, Seattle, Washington, Oct. 1996.
- [9] R. J. Fowler. *Decentralized Object Finding Using Forwarding Addresses*. PhD thesis, University of Washington, Dec. 1985. Also available as Technical Report UW-CSE-85-12-01.
- [10] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996.
- [11] R. Grimm and B. Bershad. Future directions: System support for pervasive applications. In *Proceedings of FuDiCo 2002: International Workshop on Future Directions in Distributed Computing*, Bertinoro, Italy, June 2002.
- [12] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. Programming for pervasive computing environments. Submitted for publication, Jan. 2002.
- [13] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G. Borriello, and D. Wetherall. Programming for pervasive computing environments. Technical Report UW-CSE-01-06-01, University of Washington, June 2001.
- [14] M. Guzdial and K. Rose, editors. *Squeak: Open Personal Computing and Multimedia*. Prentice Hall, 2002.
- [15] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb. 1988.

- [16] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.
- [17] P. J. Leach and R. Salz. UUIDs and GUIDs. Internet Draft draft-leach-uuids-guids-01.txt, Internet Engineering Task Force, Feb. 1998.
- [18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
- [19] D. Milojević, F. Douglass, and R. Wheeler, editors. *Mobility—Processes, Computers, and Agents*. ACM Press. Addison-Wesley, Feb. 1999.
- [20] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track, 1999 USENIX Annual Technical Conference*, pages 183–192, Monterey, California, June 1999.
- [21] M. L. Powell and B. P. Miller. Process migration in DEMOS/MP. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 110–119, Bretton Woods, New Hampshire, Oct. 1983.
- [22] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, Mar. 2001.
- [23] B. Steensgaard and E. Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 68–77, Copper Mountain Resort, Colorado, Dec. 1995.
- [24] T. Thai and H. Lam. *.NET Framework Essentials*. O’Reilly, 2nd edition, Feb. 2002.
- [25] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Proceedings of the 8th ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, Sept. 1998.
- [26] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, Sept. 1991.
- [27] J. E. White. Mobile agents. In Milojević et al. [19], pages 460–493.