Programming for Pervasive Computing Environments

ROBERT GRIMM, JANET DAVIS, ERIC LEMAR, ADAM MACBETH,
STEVEN SWANSON, TOM 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. However, 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 introduce a system architecture for pervasive computing, called *one.world*. Our architecture provides an integrated and comprehensive framework for building pervasive applications. It includes a set of services, such as service discovery, checkpointing, and migration, that help to build applications and directly simplify the task of coping with constant change. We describe the design and implementation of our architecture and present the results of building applications within it. Our evaluation demonstrates that by using *one.world* programming for highly dynamic computing environments becomes tractable and that measured programmer productivity does not decrease when compared to more conventional programming styles.

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

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

1. INTRODUCTION

In this paper, we explore how to build applications for pervasive computing environments. Pervasive, or ubiquitous, computing [Esler et al. 1999; Weiser 1991] calls for the deployment of a wide variety of smart devices throughout our working and living spaces. These devices are intended to react to their environment and coordinate with each other and with network services. Furthermore, many devices will be mobile. They are expected to dynamically discover other devices at a given location and continue to function even if they are disconnected. The overall goal is to provide users with universal and immediate access to information and to trans-

Authors' address: Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195, USA.
This work was funded in part under DARPA contracts F30602-98-1-0205 and N66001-99-2-892401. Davis and Swanson were partially supported by NSF graduate fellowships. Davis, Grimm, and Lemar were partially supported by Intel Corporation internships. Grimm was also supported by IBM Corporation and Intel Foundation graduate fellowships.

parently support them in their tasks. The pervasive computing space can thus be envisioned as a combination of mobile and stationary devices that draw on powerful services embedded in the network to achieve users' tasks [Dertouzos 1999]. The result will be a giant, ad-hoc distributed system, with tens of thousands of people, devices, and services coming and going.

The key challenge for developers is to build applications that adapt to such a highly dynamic environment and continue to function even if people and devices are roaming across the infrastructure and if the network provides only limited services. However, existing approaches to building distributed applications, including client/server or multitier computing, are ill-suited to meet this challenge. They are targeted at less dynamic computing environments and lack sufficient facilities for managing constant change. As a result, developers of pervasive applications have to expend considerable effort towards building necessary systems infrastructure rather than focusing on actual applications.

To mitigate this situation, we introduce a system architecture for pervasive computing, called *one.world*. Our architecture is based on a simple programming model and provides a set of services that have been specifically designed for large and dynamic computing networks. Our architecture does not introduce fundamentally new operating system technologies or services; rather, the goal is to provide an integrated and comprehensive framework for building pervasive applications. By using *one.world*, application developers can focus on application logic and on making applications adaptable. We have validated our approach by building and evaluating two applications, Emcee and Chat. Emcee manages users and their applications, including the ability to move or copy applications between users and to move all of a user's applications between machines, while Chat provides text and audio messaging.

Together, Emcee and Chat provide the applications for a simple pervasive computing scenario in which members of a group, such as a research group, access their applications and communicate with each other, independent of their current location. Group members have their own offices, but frequently move throughout their organization's space and spend significant amounts of time in shared areas, such as laboratories or meeting rooms. They use mobile devices to run their applications, but they also use the computers in their work environment. As they move through the organization's space, they exchange text messages, listen to music, or talk with each other.

In contrast to existing networked application support, such as X Windows [Nye 1995] or roaming profiles in Windows [Tulloch 2001], our scenario requires that setup and maintenance be minimal and that application access be seamless and ubiquitous. In particular, our scenario does not rely on all-powerful administrators and well-maintained servers, locked up in a machine room. In fact, our group may be an ad-hoc working group formed at a conference. Furthermore, users do not need to manually restart their applications when moving between machines; rather, they simply resume where they left off. Finally, visitors can easily join the group by starting their own applications or by copying another user's applications.

The contributions of this paper are threefold. First, we discuss the shortcomings of existing distributed systems technologies when applied to pervasive computing.

These technologies typically extend single-node programming methodologies and hide distribution, making them unsuitable for highly dynamic computing environments. As an alternative, we present a principled approach that cleanly exposes distribution and is more appropriate for this space. Second, we provide a detailed description of a system architecture that embodies our approach. We describe its design and highlight interesting aspects of its implementation. Third, we demonstrate that by using *one.world* programming for change becomes tractable and that measured programmer productivity does not decrease when compared to more conventional programming styles. We also relay lessons learned and identify opportunities for future research.

This paper 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. We describe its design in Section 4 and our Java-based implementation in Section 5. Section 6 presents the two applications we have built and explores the impact of writing adaptable applications. We present an evaluation of our architecture in Section 7, quantifying both developer effort and application performance. In Section 8, we reflect on our experiences with building and using *one.world*. Section 9 reviews related work. Finally, Section 10 concludes this paper.

2. MOTIVATION AND APPROACH

From a systems viewpoint, the pervasive computing space presents the unique challenge of a large and highly dynamic distributed computing environment. This suggests that pervasive applications really are distributed applications. Yet, existing approaches to building distributed systems do not provide adequate support for pervasive applications 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, this transparency 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 local and remote access, 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 tens of thousands of devices and services come and go, change is inherent and the unavailability of some resource is a frequent occurrence.

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 programming paradigm for single-node applications to distributed systems. However, by encapsulating data behind an object's interface, objects limit how data can be used and 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, it requires connected operation for any use other than reading static pages. Second, it places the burden of adapting to change on users, for example, by making them reload a page when a server is unavailable. Finally, it does not seem to accommodate emerging technologies that are clearly useful for building adaptable applications, such as mobile code [Thorn 1997] (beyond its use for enlivening pages) and service discovery [Adjie-Winoto et al. 1999; Arnold et al. 1999; Czerwinski et al. 1999].

This raises the question of how to structure systems 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 provide a better alternative, we identify three principles that should guide the design of a systems framework for pervasive computing.

PRINCIPLE 1. *Expose change.*

Systems should expose change, including failures, rather than hide distribution, so

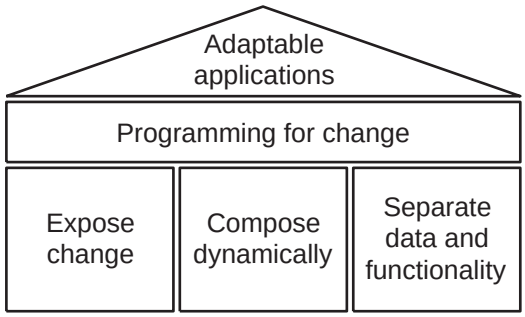

Fig. 1. Overview of our approach. The three principles guide the design of our system architecture and make it feasible for application developers to program for change, resulting in adaptable applications.

that applications can implement their own strategies for handling changes. 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 he or she moves through the physical world, and "find matching resource" to discover suitable resources on the network, such as other users to exchange messages with.

Principle 2. *Compose dynamically.*

Systems should make it easy to compose and expand applications and services at runtime. In particular, interposing on a component's interactions with other components or the outside world must be simple. Such features make 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.

Principle 3. *Separate data and functionality.*

Systems need to provide a clean separation between data and functionality, so that they can be managed separately and so that they can evolve independently. The separation is especially important for services that search, filter, or translate large amounts of data. At the same time, data and functionality depend on each other, for example, when migrating our Chat application and the music it is currently broadcasting. Systems thus need to include the ability to group data and functionality but must make them accessible independently.

Common to all three principles 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 more traditional distributed systems.

At the same time, the three principles do not preclude the use of established programming methodologies. Exposing change does not prevent us from providing reasonable default behaviors. But it does emphasize that applications must be notified of change. Similarly, composing dynamically does not preclude the use of strong typing, for example, in the form of strongly typed events. However, it does emphasize the need for simplifying interposition. Separating data and functionality does not preclude the use of object-oriented programming. The ability to abstract data or functionality is clearly useful for structuring and implementing applications. Rather, separating data and functionality emphasizes that application data and functionality should build on distinct abstractions.

More importantly, a system architecture whose design follows the three principles provides considerable support for dealing with change. Exposing change helps with identifying and reacting to changes in devices and the network. Dynamic composition helps with changes in application features and behaviors. Finally, separating data and functionality helps with changes in data formats and application functionality. Given a system that follows these principles, application developers can focus on making applications adaptable instead of creating necessary systems support. This approach to building pervasive applications is illustrated in Figure 1.

With the principles in place, we now provide an overview of our architecture and introduce the basic abstractions as well as the core services.

3. ARCHITECTURE

In our architecture, each device typically runs a single instance of *one.world*. Each such *node* is independent of other nodes and need not be connected with other nodes. Furthermore, each node may be administered separately. Applications run within *one.world*, and all applications running on the same node share the same instance of our architecture. Our architecture provides the same basic abstractions and core services across all nodes and uses mobile code to provide a uniform and safe execution platform.

3.1 Basic Abstractions

Our architecture relies on separate abstractions for application data and functionality. Applications store and communicate data in the form of *tuples* and are composed from *components*. Tuples define a common data model, including a type system, for all applications and thus make it easy to store and exchange data. They are records with named fields and are self-describing in that an application can dynamically determine a tuple's fields and their types. Components implement functionality and interact by exchanging asynchronous events through imported and exported event handlers. All event handlers implement the same, uniform interface, making it easy to compose them. Imported and exported event handlers are dynamically added to and removed from a component and are dynamically linked and unlinked.

Environments provide structure and control. They serve as containers for tuples, components, and other environments and form a hierarchy with a single root per node. Each application has at least one environment, in which it stores tuples and in which its components are instantiated. However, applications are not limited to a single environment and may span several, nested environments. Each application

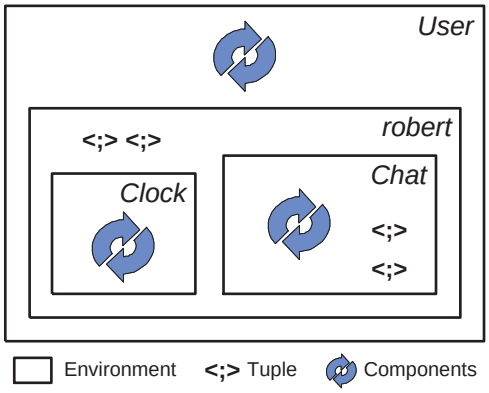

Fig. 2. An example environment hierarchy. The "User" environment hosts the Emcee application and has one child, named "robert", which stores tuples. The *robert* environment in turn has two children, named "Clock" and "Chat". The *Clock* environment only contains active components, while the *Chat* environment, in addition to hosting the Chat application, also stores tuples.

runs within its own protection domain, isolating applications from each other and from *one.world*'s kernel, which is hosted by each node's root environment. Applications are notified of important events by their environments, including that the environment has been activated, restored from a checkpoint, or migrated. Environments are also 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 by file system directories and nested processes [Brinch Hansen 1970; Ford et al. 1996; Tullmann and Lepreau 1998] in other operating systems, while still preserving independent access to data and functionality. Figure 2 shows an example environment hierarchy.

3.2 Services

In addition to the basic abstractions, *one.world* provides a set of services that serve as common building blocks and help developers in making their applications adaptable. *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.

Migration provides the ability to move or copy an environment and its contents, including tuples, components, and nested environments, either locally or to another node. It is especially useful for applications that follow a user from shared device to shared device as he or she moves through the physical world. *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

Table I. Overview of *one.world*'s features. *Issue* specifies the distributed systems concern. *Feature* describes the corresponding *one.world* abstraction or service. *Section* lists the design or implementation section discussing that feature.

Issue	**Feature**	**Section**
Application model	Applications are composed from components that exchange events.	4.2
	Operations manage event exchanges, notably those with event handlers outside an application.	4.2.1
Process management	Environments contain application components.	4.3
	Checkpointing captures an application's execution state and migration moves or copies an application.	4.3, 5.3
Addressability / Naming	Protection domains limit access to direct references.	5.1
	The environment hierarchy limits access to nested environments.	4.3
	Remote event passing and discovery provide access to arbitrary event handlers through symbolic handlers.	4.2.2, 5.2
Storage	Structured I/O persistently stores tuples in environments.	4.1
Communications	Remote event passing and discovery send events to remote receivers.	4.2.2, 5.2
Security	Protection domains isolate applications.	5.1
	The request/monitor mechanism can be used to implement reference monitors and auditing.	4.3
Resource allocation	The request/monitor mechanism can be used to interpose on requests for system services.	4.3
Extensibility	The request/monitor mechanism can be used to add new services.	4.3

or after a failure, such as a device's batteries running out.

Remote event passing (REP) provides the ability to send events to remote services and is *one.world*'s basic mechanism for communication across the network. To use REP, services export event handlers under symbolic descriptors, that is, tuples, and clients send events by specifying the symbolic receiver. Finally, *discovery* routes events to services with unknown location. It supports a rich set of options, including early and late binding [Adjie-Winoto et al. 1999] as well as 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 discover local resources, such as a wall display or printer.

As a distributed systems architecture, *one.world* must address several distributed systems issues. Table I lists the most important issues and relates them to the corresponding features in our architecture. It also lists the specific sections that discuss these features in detail, thus serving as an index into the design and implementation sections of this paper.

4. DESIGN

We now describe the design of *one.world* in detail, explaining our design decisions and illustrating the use of our architecture through code examples. In contrast to the previous section, which distinguishes between basic abstractions and core

```
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 List   fields() {...}
  public final Class  getType(String name) {...}

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

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

Fig. 3. Definition of a tuple. All tuples inherit this base class and have an ID field to support symbolic references and a metadata field to support application-specific annotations. They also have a set of methods to programmatically access a tuple's fields, to validate a tuple's semantic constraints, and to convert the tuple into a human-readable representation. A `Guid` is a globally unique identifier. A `DynamicTuple` is a special tuple; its fields can be of any type and, unlike those of other tuples, can be dynamically added and removed. The accessor methods are final and are implemented using reflection. In contrast, individual tuple classes can override the `validate()` and `toString()` methods to define their own semantic constraints and human-readable representation, respectively.

services, this section is structured according to our principle of separating data and functionality. We first present our architecture's facilities for data management in Section 4.1, followed by event processing in Section 4.2 and the environment hierarchy in Section 4.3. *one.world*'s core services are discussed in the appropriate subsections: operations, remote event passing, and discovery in Section 4.2; checkpointing and migration in Section 4.3.

4.1 Data Management

Data management in *one.world* is based on tuples. Tuples define the common data model for all applications running in our architecture, thus simplifying the sharing of data. They are self-describing, mutable records with named and optionally 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 form of a *box*, which contains a serialized representation of the object. All tuples share the same base class and have an ID field specifying a globally unique identifier [Leach and Salz 1998] (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, access its data, validate its semantic constraints (for example, to determine whether a tuple's field values are consistent with each other), and produce a human-readable representation. The base class for all tuples is shown in Figure 3.

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

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.

Our data model also defines a common query language for tuples, which is used for both storage and service discovery. Queries support comparison of a constant to the value of a field, including the fields of nested tuples, comparison to the declared or actual type of a tuple or field, and negation, disjunction, and conjunction. Queries are expressed as tuples; an example query is shown in Figure 8.

Structured I/O lets applications store tuples in environments. Each environment's tuple storage is separate from that of other 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 II. 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, applications *bind* to tuple storage and then perform operations on the bound resource. All bindings are controlled by leases [Gray and Cheriton 1989].

We chose tuples instead of byte strings for I/O because tuples preserve the structure of application data. Tuples obviate the need for explicit marshaling and unmarshaling of data and enable system-level query processing. Since they provide well-defined data units, they also make it easier to share data between multiple writers. 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.

We chose a storage mechanism that is separate from communications, provided by remote event passing, instead of a unified tuple space abstraction [Carriero and Gelernter 1986; Davies et al. 1998; Freeman et al. 1999; Murphy et al. 2001; Wyckoff et al. 1998] because such a separation exposes distribution and change in the environment. Furthermore, it better reflects how pervasive applications store and communicate data. On one side, many applications need to modify stored data. For example, a personal information manager needs to be able to update stored contacts and appointments. Structured I/O lets applications overwrite stored tuples by simply writing a tuple with the same ID as the stored tuple. In contrast, tuple spaces only support the addition of new tuples, but existing tuples cannot be changed. On the other side, some applications, such as streaming audio and video, need to directly communicate data in a timely fashion. Remote event passing pro-

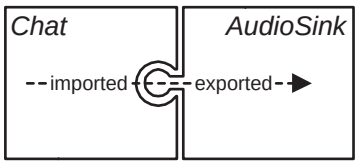

Fig. 4. Illustration of the relationship between imported and exported event handlers. Boxes represent components, indentations represent imported event handlers, and protrusions represent exported event handlers. The dotted arrow indicates the direction of event flow. In this example, the component named "Chat" imports an event handler named "imported", and the component named "AudioSink" exports an event handler named "exported". The two event handlers are linked. When an event is sent to the *imported* event handler, that is, when that event handler is invoked on an event, the event is forwarded to the *exported* event handler, which then processes it. In the case of *Chat* and *AudioSink*, *Chat* sends received audio messages to the *AudioSink*, which then plays back the audio contained in the messages.

vides that functionality. In contrast, tuple spaces store all tuples before delivering them and consequently retain them in storage. This is especially problematic for streaming audio and video, since data tends to be very 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.

4.2 Event Processing

Control flow in *one.world* is expressed through asynchronous events that are processed by event handlers. Events are represented by tuples. In addition to the ID and metadata fields common to all tuples, 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. For request/response interactions, the closure of the request is returned with the response. Closures are useful for storing additional state needed for processing responses and thus can simplify the implementation of event handlers. Event handlers implement a uniform interface with a single method that takes the event to be processed as its only argument. Event delivery has at-most-once semantics, both for local and remote event handling.

Components implement application functionality. 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. Components 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. An application's main component has a static initialization method that instantiates its components and performs the initial linking. 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. The relationship between imported and exported event handlers is

```
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. 5. 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 method shown in this figure first instantiates the Emcee component and then links that component with its environment. It links the `main` event handler imported by the environment `env` with the `main` event handler exported by the component `comp`, and the `request` event handler imported by the component `comp` with the `request` event handler exported by the environment `env`. The role of the `main` and `request` event handlers is explained in Section 4.3. Note that linked event handlers need not have the same name, although they do in this example.

illustrated in Figure 4, and an example initialization method is shown in Figure 5.

To implement asynchronous event handling, each environment provides a queue of pending ⟨*event handler, event*⟩ invocations as well as a pool of one or more threads to perform such invocations. When an event is sent between components in different environments, the corresponding event handler invocation is automatically enqueued in the ⟨*event handler, event*⟩ queue of the target environment. If the environments are in different protection domains, the event is also copied before it is enqueued. When an event is sent between components in the same environment, the event handler invocation is a direct method call, so that the event is delivered reliably and efficiently. This default can be overridden at link-time, so that event handlers in the same environment use the ⟨*event handler, event*⟩ queue instead of direct invocations.

We chose to use asynchronous events instead of synchronous invocations 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. Second, 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,

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

Fig. 6. 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 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.

4.2.1 *Operations.* While asynchronous events provide a good fit for pervasive computing, they also raise the question of how to structure applications, 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. In our experience with writing event-based code, established styles of event-based programming, such as state machines, are only manageable for very simple applications.

After some experimentation, we found the following approach, which we call the *logic/operation pattern*, particularly successful. Under this pattern, an application is partitioned into logic and operations, which are implemented by separate sets of event handlers. Logic are computations that do not fail, barring catastrophic failures, such as creating or displaying a text message. Operations are interactions that may fail, such as sending a text or audio message to its intended recipients. Operations include all necessary failure detection and recovery code. The 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.

The `Operation` service reifies the logic/operation pattern. 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 the failure. 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 to express complex interactions and structure event-based applications. Example code for creating an operation is shown in Figure 6. That operation is then used to manage the request/response interactions shown in Figures 8 and 9.

4.2.2 *Remote Event Passing and Discovery.* Besides obviously needing to communicate with each other, mobile devices and migrating applications need to be

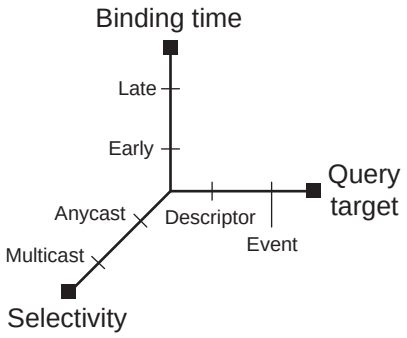

Fig. 7. A classification of discovery options. The binding time determines whether the discovery query is performed early (before sending an event) or late (while routing an event). The query target determines whether the query is performed on the service descriptor or the event itself (which represents a form of reverse lookup). The selectivity determines whether an event is sent to a single matching resource or to all matching resources. Note that not all combinations are meaningful; notably, early binding cannot be combined with the event as a query target because queries on the event itself can only be performed while routing the event.

able to locate each other and nearby resources. As a result, discovery is an important service for dynamically composing pervasive applications and needs to support a rich set of options. We classify the major discovery options along three axes. The first axis represents the binding time, which determines when to perform a discovery query. Early binding first resolves the query and then uses point-to-point communications with the resolved resource. It is useful when an application needs to repeatedly send events to the same resource or when services can be expected to remain in the same location. Late binding [Adjie-Winoto et al. 1999] resolves the query only while routing the event. While it introduces a performance overhead for every sent event, late binding also is the most responsive and thus most reliable form of communication in a highly dynamic environment. The second axis represents the query target, which determines the entity on which to perform a discovery query. While the query is typically performed on service descriptors, it can also be performed on the events themselves. The latter constitutes a form of reverse lookup and is useful for such tasks as logging and debugging remote communications. Finally, the third axis represents the specificity, which determines the number of resources receiving an event. Anycast sends the event to a single matching resource, while multicast sends the event to all matching resources. Our classification of discovery options is illustrated in Figure 7.

Remote event passing (REP) provides the ability to send events to remote receivers. It supports both point-to-point communications and service discovery, including all options described above, through only three simple operations: *export*, *send*, and *resolve*. The *export* operation makes an event handler accessible from remote nodes through a symbolic descriptor, that is, a tuple. The descriptor's type determines how the event handler is exported. If the descriptor is a `Name`, the event handler is exported for point-to-point communications. If it is a `Query`, the event handler is exported for reverse lookups on the events sent through late binding

```
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. 8. 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 known, the event is sent through point-to-point communications. 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`. 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.

discovery. For all other tuples, the event handler is exported for regular discovery lookups. The resulting binding between the event handler and descriptor is leased. The *send* operation sends an event to previously exported event handlers using either a node address and name for point-to-point communications or a discovery query for late binding. For late binding, a flag determines whether to use anycast or multicast. Finally, the *resolve* operation looks up event handlers in the discovery service so that they can be used for point-to-point communications. Example code for sending an event through REP is shown in Figure 8.

Discovery relies on a centralized server to provide its functionality. The discovery server is automatically elected from all nodes running *one.world* on the local network, with elections favoring nodes with ample resources and long uptimes. Discovery server election eliminates the need for manual configuration and administration and thus makes it possible to use discovery outside well-managed computing environments, such as a conference site. Each node automatically forwards exported $\langle$*event handler, descriptor*$\rangle$ bindings to the current discovery server, routes late binding sends through the server, and performs event handler resolutions on the server.

4.3 The Environment Hierarchy

Environments are containers for stored tuples, components, and other environments and provide structure and control in *one.world*. They provide structure by grouping data and functionality, and they provide control by nesting environments within each other. At the same time, environments always maintain a clear separation between data and functionality, which can be accessed independently and are not hidden behind a unifying interface. The environment abstraction was inspired by the ambient calculus [Cardelli 1999]. Similar to environments, ambients serve as containers for data, functionality, and other ambients. The difference between the

```
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. 9. Code examples for checkpointing, restoring, and moving an environment. The first code snippet checkpoints a user's environment `env`. The second code snippet restores the latest checkpoint for a user's environment `env`. The third code snippet moves a user's environment `user.env` to the node named `location`. For all snippets, the operation forwards the event to *one.world*'s kernel, which then performs the requested environment 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.

two abstractions is a difference of focus: ambients are used to reason about mobile computations, while environments are used to implement applications.

The grouping of data and functionality is relevant for loading code, checkpointing, and migration. In *one.world*, application code is stored as tuples and loaded from environments. Checkpointing captures the execution state of an environment tree, including application components and pending $\langle$*event handler, event*$\rangle$ invocations, in the form of a tuple that is stored in the root of the checkpointed environment tree. The environment tree can later be reverted by reading the tuple and restoring the execution state. Finally, migration provides the ability to move or copy an environment tree, including all execution state and all stored tuples, to a remote node. Migration is eager in the sense that it moves or copies the environment tree in a single operation, leaving no implicit back-references to the originating node. Example code for checkpointing, restoring, and moving an environment is shown in Figure 9.

Checkpointing and migration need to capture and restore the execution state of an environment tree. When capturing execution state, our architecture first quiesces all environments in the tree, that is, waits for all threads to return to their thread pools. It then serializes the affected application state, notably all components in the environment tree, and the corresponding environment state, notably the $\langle$*event handler, event*$\rangle$ queues. When restoring execution state, *one.world* first deserializes all application and environment state, then reactivates all threads, and finally notifies applications that they have been restored or migrated. Upon receiving this notification, applications restore access to outside resources, such as bindings for tuple storage or REP.

During serialization, all non-symbolic references to event handlers outside the affected environment tree are nulled out. This includes links to components outside the tree or event handlers providing access to kernel resources, such as structured I/O. Applications need to restore nulled out handlers themselves by relinking or

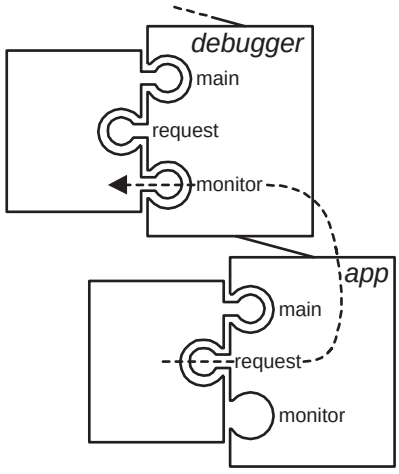

Fig. 10. Illustration of the request/monitor mechanism. Boxes on the left represent application components and boxes on the right represent environments. The *app* environment is nested within the *debugger* environment. The *debugger* environment's monitor handler is linked and thus intercepts all events sent to the *app* environment's request handler. The use of the main handler is explained in the text.

rebinding after restoration or migration. While an environment tree is being migrated, components in the tree cannot receive events from outside the tree. An attempt to send an event to a component currently being migrated results in a failure notification, thus exposing the change in component location. The component can accept events only after migration has completed and the application has restored access to outside resources again.

The nesting of environments is relevant for the following two features. First, logic to control checkpointing and migration can be separated into an outer environment, because checkpointing and migration affect an entire environment tree. For example, a migration agent that knows how to follow a user as he or she moves through the physical world can migrate *any* application, simply by dynamically embedding the application in its environment. Second, interposition gives an outer environment complete control over an inner environment's interactions with environments higher up the hierarchy, including *one.world*'s kernel.

To an application, an environment appears to be a regular component, even though the environment's event handlers belong to the root environment's protection domain. Each environment imports an event handler called "main", which must be linked to an application's main component before the application can run in the environment. It is used by *one.world* to notify the application of important events, such as activation, restoration, migration, or termination of the environment.

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 structured I/O, REP, and environment operations. Consequently, applications use the request handler for interacting with the kernel. For example, the operation created in Figure 6 and used in Figures 8 and 9 forwards events to the request handler of Emcee's environment. 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 and by linking to its monitor handler. This use of the *request/monitor mechanism* is illustrated in Figure 10.

To enforce the nesting of environments, *one.world* restricts access to tuple storage and environment operations, such as creating or deleting an environment, to the requesting environment and its descendants. When an application sends an event to its request handler, the event's metadata is tagged with the identity of the requesting environment. Before granting access to tuple storage or performing an operation on an environment, the kernel verifies that the requesting environment is an ancestor of the environment being operated on.

We chose a hierarchical arrangement for environments because, while conceptually simple, it offers considerable flexibility and power. In particular, the request/monitor mechanism—by building on our component model—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 3, 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.

We chose to null out references to outside event handlers during serialization because doing so exposes change in environments. For migration, an alternative approach might redirect such event handlers to the original node. However, transparently redirecting event handlers creates residual dependencies [Powell and Miller 1983] and thus increases an application's exposure to failures, while also hiding the cause of failures from the application. Furthermore, nulling out event handlers typically does not place an additional burden on developers, because applications already need to explicitly acquire resources at other points in their life cycles, such as when they are activated.

5. IMPLEMENTATION

In this section, we present *one.world*'s implementation. After a short overview of our implementation, we explore the more interesting aspects in detail. We present how environments are separated from each other and how events are passed between them in Section 5.1. We then discuss the implementation of remote event passing and discovery in Section 5.2. Finally, we describe the implementation of checkpointing and migration in Section 5.3.

The implementation of *one.world* currently 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] to implement reliable tuple storage. The implementation currently lacks support for structured I/O transactions and for loading code from environments, although their implementation should be straightforward. It does, however, include support for building GUI-based applications, for a command line shell, and for converting between files and stored tuples.

The implementation of *one.world* has approximately 19,000 non-commenting source statements (NCSS). Our entire source tree, including regression tests, benchmarks, and applications, has approximately 46,000 NCSS. 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, support for reflection and object serialization, and the ability to customize the code loading process. As a result, *one.world* could also be implemented on other platforms that provide these features, such as Microsoft's .NET [Thai and Lam 2001].

5.1 Environment Separation

In *one.world*, each environment processes events independently from other environments by using its own queue of pending $\langle$*event handler, event*$\rangle$ invocations and thread pool, and each application has its own protection domain. Events sent across environments are automatically placed into the appropriate $\langle$*event handler, event*$\rangle$ queue and are also copied if they are sent across protection domain boundaries. Our implementation uses two related mechanisms, class loaders and event handler wrapping, to provide this separation of environments.

Class loaders are used to separate the code of applications in different protection domains. They are arranged in a simple hierarchy [Liang and Bracha 1998], with one root class loader having a child class loader for every application's protection domain. The root class loader is responsible for loading core classes, including *one.world*'s kernel, and is created on startup. Child class loaders load an application's code, but defer to the root class loader for Java's platform classes and *one.world*'s core classes. They are created when an application is created or when it migrates to a node and are destroyed when the application is destroyed or when it moves to another node. As a result, core classes are shared between protection domains and loaded once, and application code only needs to be in memory while an application is running.

Event handler wrapping ensures that events are automatically placed into the appropriate $\langle$*event handler, event*$\rangle$ queue and that arbitrary object references cannot be leaked between protection domains. The basic idea behind event handler wrapping is that an application cannot directly reference an event handler originating from another environment, but only a wrapped version [Shapiro 1986]. An application initially accesses event handlers in other environments by linking with event handlers exported by components in other environments, such as its environment's request handler which is implemented by the root environment. The linker performs the event handler wrapping as part of the linking process. Any additional

wrapping is then performed by the wrapped event handlers themselves.

A wrapped event handler keeps internal references to the original, unwrapped event handler, the unwrapped event handler's environment (which we call the target environment because it receives events), and the environment using the wrapped handler (which we call the source environment because it sends events). When an event is sent to a wrapped event handler, the wrapped event handler first ensures that all event handlers referenced by the event are correctly wrapped. By using the tuple accessor methods shown in Figure 3, it traverses all fields of the event, including the fields of nested tuples, and modifies event handlers in place as necessary. If the protection domains of the source and target environments differ, the wrapped event handler then copies the event. Events whose classes were loaded by the root class loader are simply copied. Events whose classes were loaded by a child class loader are recreated using the child class loader of the target environment's protection domain. After wrapping and, if necessary, copying the event, the wrapped event handler enqueues the unwrapped event handler and the resulting event in the target environment's $\langle$*event handler, event*$\rangle$ queue.

Taken together, class loaders and event handler wrapping separate environments from each other and provide a well-defined method for communicating between them. Copying events and wrapping event handlers is sufficient to prevent protection domain leaks because tuples and event handlers are the only entities applications can define themselves. All other types that can be used in tuples are defined by our architecture and cannot reference arbitrary objects. Note that wrapping event handlers in place before copying events does not represent a security hole: While an application might be able to replace an already wrapped event handler with an unwrapped one, the event copy code only copies references to wrapped event handlers, but never unwrapped event handlers.

5.2 Remote Event Passing and Discovery

Remote event passing provides the ability to symbolically name event handlers and is the only way to remotely reference event handlers in *one.world*, as our architecture does not support direct references to remote event handlers. The event handler descriptors used by REP, such as the `NamedResource` and `DiscoveredResource` shown in Figure 8, provide a layer of indirection to actual event handlers. They nominally implement the event handler interface, so that they can be used instead of actual event handler references. However, sending an event to such a *symbolic handler* always results in a failure condition. Furthermore, in keeping with a design that makes remote communications explicit, events that are sent remotely may only reference symbolic handlers but not actual handlers. REP does not automatically map event handlers to symbolic handlers, and applications have full control over which event handlers they expose for remote communications.

To implement point-to-point communications, REP uses a table mapping the names encapsulated by `NamedResources` to the actual event handlers. Mappings are added through the *export* operation and removed when the corresponding leases are canceled or expire. To send an event through point-to-point communications, REP forwards the event to the node specified by the `NamedResource`, where the name is resolved to an event handler by performing a table lookup and the event is delivered to that event handler. The implementation uses a communications

substrate that sends arbitrary tuples across UDP or TCP by using Java's object serialization. REP defaults to TCP, but senders can override this default when sending an event. The choice of UDP or TCP affects the reliability and timeliness of event delivery between nodes, but has no other application-visible effect. For TCP-based communications, REP maintains a cache of connections to avoid re-creating connections. Furthermore, sending events across the network is avoided altogether if the sender and the receiver are on the same node.

Discovery is implemented on top of REP's point-to-point communications and is split into three components: a discovery client, a discovery server, and an election manager. The discovery client and the election manager run on every node, while the server usually runs on only one node for a local network. The discovery client is responsible for maintaining the discovery bindings for all applications running on a node and for forwarding discovery requests to the discovery server. To maintain discovery bindings, the discovery client uses an internal table. Just as for point-to-point communications, bindings are added through the *export* operation and removed when the corresponding leases are canceled or expire. When a discovery server becomes visible on the local network, the discovery client propagates all bindings to the server. Server-side bindings are leased and the discovery client maintains these internal leases. To forward discovery requests, the discovery client simply sends the requests to one of the currently visible servers.

The discovery server is responsible for actually servicing discovery requests. It accepts the bindings propagated by discovery clients and integrates them into a single table for all applications on the local network. Descriptors that are identical, ignoring tuple IDs and metadata, are collapsed into a single table entry to improve the performance of query processing. When processing a *resolve* operation, the discovery server looks up matching services and returns the result to the discovery client that forwarded the operation. When processing a late binding send, the discovery server first processes reverse lookups on the event (which do not count as a match for anycast) and then forward lookups on the resource descriptors. The event is then forwarded directly to the matching service for anycast and to all matching services for multicast. If no service matches, a failure notification is reflected back to the sending application.

The election manager is responsible for ensuring that a discovery server is present on the local network. The current discovery server periodically announces its presence, every two seconds in our implementation. Announcements are sent as UDP multicasts through the tuple-based communications substrate. The election manager listens for discovery server announcements. If it does not receive any announcements for two announcement periods, it calls an election. During an election, each node broadcasts a score, computed from a node's uptime and memory size. The node with the highest score wins the election and starts the discovery server.

Also, the discovery client calls an election when it receives a malformed or unexpected event indicating that the current discovery server has failed. Furthermore, the discovery server calls an election when its node is about to be shut down. If two discovery servers run on the same network due to a changing network topology, the server with the lower score shuts down when it sees an announcement from the server with the higher score. Discovery still works with two servers, since bindings

are propagated to all visible servers and requests are forwarded to a single server. Conversely, in case of a network partition, the partition without the discovery server simply elects its own server.

Since *one.world* provides a discovery *service*, our server-based implementation of discovery is largely transparent to applications. The exception occurs during a transition between servers. In that case, there is a period when not all services may be visible, because discovery bindings are still being propagated to the new discovery server. This transitional period could usually be hidden by using more than one discovery server, which could also improve the scalability of our discovery service by load balancing requests across several servers.

5.3 Checkpointing and Migration

At the core of checkpointing and migration lies the ability to capture and restore the execution state of an environment tree. Our implementation builds on Java's object serialization. When capturing the execution state of an environment tree, our implementation first quiesces all environments in the tree by waiting for the environments' threads to complete event processing. It then serializes each environment's main and monitor handlers; all application components must be reachable from these two event handlers. It also serializes each environment's ⟨*event handler, event*⟩ queue. During serialization, our implementation nulls out wrapped event handlers whose target environment is not in the environment tree, with the exception of request handlers belonging to environments in the tree, which are preserved. Furthermore, Java classes are annotated with their protection domain. When deserializing a Java class, the class can then be loaded by the appropriate class loader. A checkpoint is represented as a `CheckPoint` tuple, which contains the resulting binary data, the identifiers for the environments in the checkpoint, and a timestamp.

The implementation of checkpointing uses structured I/O to read and write `CheckPoint` tuples. After creating a checkpoint, it writes the checkpoint to the root of the checkpointed environment tree by using a structured I/O *put* operation. When restoring a checkpoint, our implementation can either restore a checkpoint with a specific timestamp or the latest checkpoint. A specific checkpoint is read using a structured I/O *read* operation that queries for a `CheckPoint` tuple with the specified timestamp. The latest checkpoint is read by using a structured I/O *query* operation and then iterating over all `CheckPoint` tuples to determine the latest checkpoint.

The implementation of migration uses a multi-round protocol on top of REP's point-to-point communications. Both sender and receiver use operations to manage sent events: the sender's operation connects each request to its response, while the receiver's operation connects each response to the next request. The sending node first seeks permission to migrate an environment tree to the new parent environment on the receiving node. After receiving permission, it sends the metadata for the environment tree, including each environment's name and parent environment. Next, it sends all tuples stored in the environment tree, sending one tuple per protocol round. Stored tuples are not deserialized and serialized during migration; rather, their binary representation is sent directly in the form of `BinaryData` tuples. Finally, the sending node sends the `CheckPoint` tuple for the environment tree. The acknowledgement for this last request completes the migration protocol.

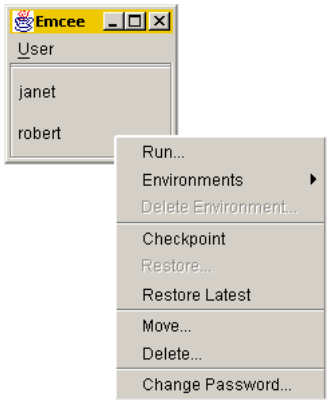

Fig. 11. Emcee's user interface. The main window lists the users whose applications run on the node. A popup menu for each user, shown for the user named "robert", is used to perform most operations, such as running a new application or checkpointing a user's applications. The user menu supports the creation of new users and the fetching of a user's applications from another node.

An error at any time on either the sender's or the receiver's side aborts the migration protocol, and a failure notification is sent to the application that initiated the migration.

The initial migration protocol request is subject to the request/monitor mechanism on the receiving node. Starting with the new parent environment, an `AcceptRequest` that describes the migrating environment tree and the sending node is delivered to the first environment whose monitor handler is linked. The migration protocol continues only if the `AcceptRequest` reaches *one.world*'s kernel. As a result, applications running in environments that would contain the migrating environment tree can redirect the migrating tree to a different parent environment by modifying the `AcceptRequest` or reject the migrating tree by dropping the `AcceptRequest`.

6. PROGRAMMING FOR CHANGE

To evaluate our architecture, we built two applications, Emcee and Chat. In this section, we introduce the two applications, describing their functionality and their implementation, and explore how programming for a highly dynamic environment has affected their structure. We follow with the results of our experimental evaluation in Section 7, where we provide a break down of the time spent developing these applications and quantify their performance.

Emcee, whose user interface is shown in Figure 11, manages users and their applications. It includes support for creating new users, running applications for a user, and checkpointing all of a user's 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 12, and dropping it onto a user's name in the main window. Finally, it supports moving all of a user's applications between nodes. Applications can either be pushed from the current node to another node, or they can be pulled from another node to the current node. Emcee can

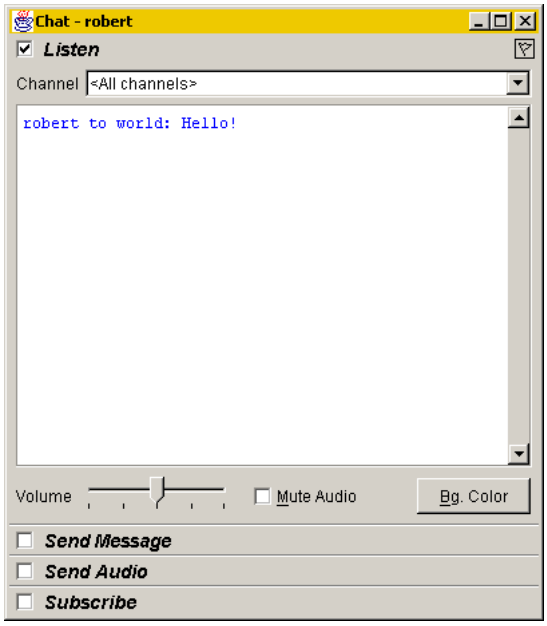

Fig. 12. Chat's user interface. The user interface is divided into four panels, which can be independently expanded or collapsed by checking or unchecking the corresponding checkbox. The *listen* panel shows received text messages and provides volume controls for audio playback. The *send message* panel 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 he or she is currently listening to.

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.

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 his or her password, and application checkpoints. The implementation of most operations is straight-forward, since they directly utilize *one.world*'s primitives (as illustrated in Figure 9). The exception is fetching a user's applications from a remote node. It uses a two-round protocol to authenticate the user to the remote instance of Emcee that is 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 node. 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 node (see Figure 8).

Chat, whose user interface is shown in Figure 12, provides text and audio messaging. It is based on a simple model in 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-

based communications for text messages and UDP-based communications 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.

The general theme for developing pervasive applications is that "no application is an island". Applications need to assume that their runtime environment changes quite frequently and that external resources are not static. Furthermore, they need to assume that their runtime environment 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 11).

For Chat, these assumptions show up throughout the implementation, with Chat verifying that its internal configuration state is consistent with its runtime environment. 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 data so that it can resume playback when migrating to a different node. It also silences a channel if the audio tuples have been deleted from their environment. Finally, before processing any event, including text and audio messages, it checks for concurrent termination.

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 detect that the user's applications have arrived on the local node. 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.

Emcee and Chat also illustrate the power of our architecture's features, specifically migration and 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 roam across the network. 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.

In addition to our experiences with Emcee and Chat, we have limited evidence that local storage can also help in building adaptable applications. Relatively early in our implementation effort we conducted an experimental comparison of *one.world* with other distributed systems technologies by teaching a senior-level undergraduate project course. The nine students in the class split into two teams that each implemented a distributed application. Each team, in turn, split into two subteams, with one subteam using existing Java-based technologies and the other using *one.world*. Since both subteams implemented the same application, this experiment lets us compare our architecture with other approaches to building distributed systems. Results are based on weekly meetings with the teams, end-of-term interviews, and the teams' final presentations and reports.

One team developed a universal inbox, which integrates a home network of future smart appliances, such as an intelligent fridge, with email access from outside the network. The universal inbox lets users access human-readable email, routes control messages to and from appliances, and provides a common data repository for email and appliance configuration state. The goal was to build a reliable application that gracefully reacts to changes in the runtime environment, such as a computer crash. The Java version uses Jini [Arnold et al. 1999] for service configuration and T Spaces [Wyckoff et al. 1998] for storing repository data. The students found the process of writing, debugging, and configuring Jini services to be a relatively arduous process. As a result, the completed implementation has relatively few services, including a centralized data repository, and each of these services represents a single point of failure. In contrast, the students using *one.world* found our architecture's support for local tuple storage and late binding discovery well matched to their needs. Their implementation avoids a centralized data repository and separates each user's email management into an independently running service. As a result, the implementation running on our architecture is more resilient to failures and more adaptable.

7. EXPERIMENTAL EVALUATION

Our experimental evaluation of *one.world* is based on Emcee and Chat. We explore whether programming for change is harder than more conventional programming styles by determining the impact programming for change has on programmer productivity. We also explore whether our implementation performs well enough to support real applications by determining how migration and discovery scale under increasing load and how applications react to a changing runtime environment. In summary, our evaluation shows that programmer productivity is within the range of results reported in the literature and is no worse than with more conventional programming styles. Furthermore, our evaluation shows that our implementation

Table III. Breakdown of development times in hours for Emcee and Chat. The reported times are the result of three authors implementing the two applications over a three month period. The activities are discussed in the text.

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

performs well enough for streaming audio between several users, and applications react quickly enough to change that users perceive only short service interruptions, if any.

To determine programmer productivity, we tracked the time spent implementing Emcee and Chat. The two applications were implemented by three authors 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 III. *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.[1] As illustrated in Section 6, *one.world* is effective at making programming for change tractable. In fact, adding audio messaging, not reacting to changes in the runtime environment, 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 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 projects [Prechelt 2000], but is almost twice as large as the results reported for a commercial company [Ferguson et al. 1999]. We thus extrapolate that programming

[1]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.

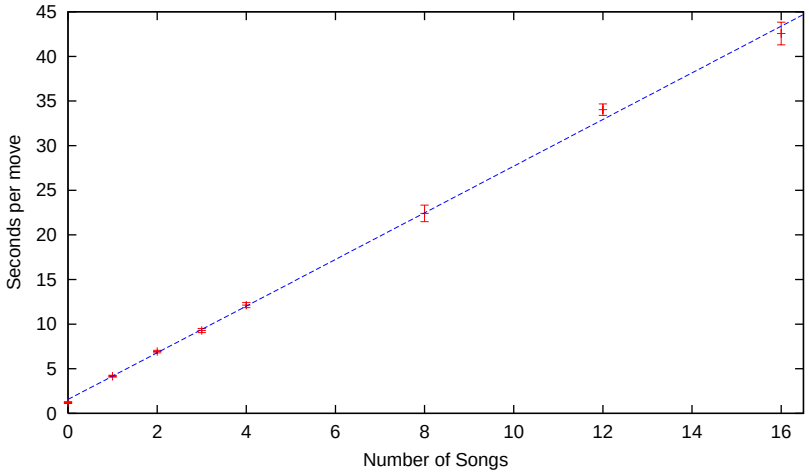

Fig. 13. Migration latency in seconds. This graph shows the latency of migrating Chat with an embedded environment containing songs. Each song has 4 MB of audio data, corresponding to 41 audio tuples. The reported numbers are the average of 60 moves, with error bars indicating the standard deviation and the line representing a least squares fit.

for change does not decrease overall programmer productivity when compared to more conventional programming styles.

To determine whether our implementation performs well enough for real application usage, we measured the scalability of migration and late binding discovery, which are the two services Emcee and Chat rely on the most, and explored how Chat reacts to a changing runtime environment. All measurements 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.

Figure 13 illustrates how migration scales under increasing load. It shows the latency for moving Chat with an embedded environment containing songs. Each song has 4 MB of audio data, corresponding to 40 stored audio tuples with 100 KB of data and one audio tuple with 20 KB of data. To perform this experiment, we used a small application that contains the Chat application and moves itself across a set of nodes in a tight loop. The results shown are the average latency in seconds for a single move, with the mover application circling 20 times around three nodes. As illustrated, migration latency grows almost linearly with the number of songs, with slight variations for 12 and 16 songs. Migration utilizes 9% of the theoretically available bandwidth and is limited by how fast stored tuples can be moved from one node 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.

Figures 14 and 15 illustrate how late binding discovery scales under increasing load. They show discovery server throughput under an increasing number of

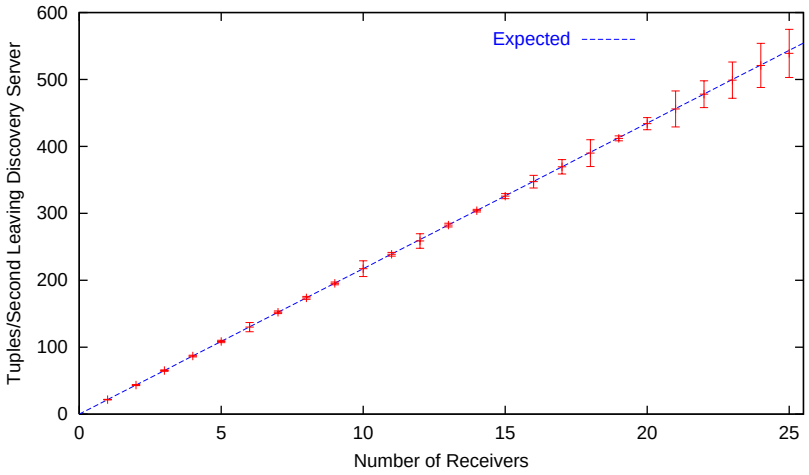

Fig. 14. Discovery server throughput under an increasing number of receivers. Throughput is measured as the number of audio messages leaving the discovery server. The results shown are the average of 30 measurements, with error bars indicating the standard deviation. Each audio message carries 8 KB of audio data.

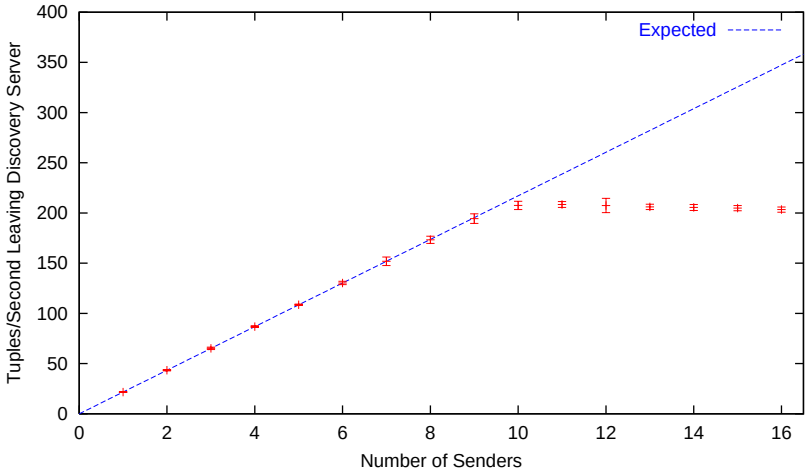

Fig. 15. Discovery server throughput under an increasing number of senders. As in Figure 14, throughput is measured as the number of audio messages leaving the discovery server. The results shown are the average of 30 measurements, with error bars indicating the standard deviation. Each audio message carries 8 KB of audio data.

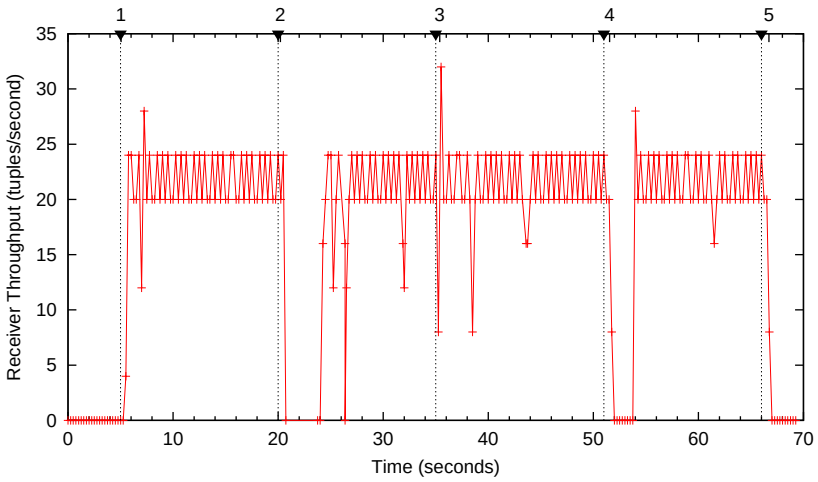

Fig. 16. Audio messages received by Chat in a changing runtime environment. Chat is subscribed to an audio channel at point 1. It is then moved to a different node at point 2. The node hosting the discovery server is shut down gracefully at point 3 and forcibly crashed at point 4. The audio channel is unsubscribed at point 5.

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 4 nodes; 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 10 senders and slightly degrades thereafter. At 10 senders, the node 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 node's performance.

Figure 16 illustrates application availability by showing the audio messages received by Chat as its runtime environment 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 experiment, 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 node and does not receive audio messages for 3.7 seconds while it migrates, re-initializes audio, and re-registers with discovery. After it has been migrated and its receiving event handler has been re-

exported to discovery, it starts receiving audio messages again. The node running the discovery server is gracefully shut down at point 3. Since that node proactively calls a discovery server election, the stream of audio messages is not interrupted. By contrast, at point 4, the node 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 announcements 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. 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. Furthermore, 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, as already suggested in Section 5.2, could improve the scalability of our discovery service and would also eliminate service interruption due to forced server elections.

8. DISCUSSION

As discussed in the previous two sections, Emcee and Chat have served as a basis for exploring the impact of programming for change and for performing a quantitative evaluation of our architecture. The process of developing these applications has also helped us gain a better understanding of the strengths and limitations of our design. We focus on the resulting insights in this section and identify lessons that are applicable beyond our work as well as opportunities for future research into pervasive computing.

Emcee and Chat make extensive use of *one.world*'s core features and illustrate the power of a design that follows the three principles presented in Section 2:

(1) *Expose change.* Event-based notification cleanly exposes change to applications. It lets us, for example, automatically adjust Chat's configuration when the user who owns the application changes.

(2) *Compose dynamically.* Environment nesting and discovery make it easy to dynamically compose functionality. We used environment nesting in Emcee to control a user's applications, notably to move or copy applications between users and to move all of a user's applications between nodes. We used discovery in Emcee to locate a user's applications and in Chat to route messages to subscribed users.

(3) *Separate data and functionality.* The separation of data and functionality provides considerable flexibility when compared to systems that combine data and functionality in objects. It lets us add music to a running Chat application, simply by importing the corresponding files into Chat's environment. It also lets us improve existing audio capabilities or add support for new types of audio sources, simply 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 REP 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 at a node close to the user. On the other side, we rely on REP to let remote users 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*. Nesting provides control over applications, as illustrated by Emcee. Nesting can also be used to extend an application's functionality. For example, we have used the request/monitor mechanism to implement a replication layer that synchronizes the tuples stored in a local environment with a master node [Grimm et al. 2001]. Furthermore, nesting is attractive because it preserves 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 moved between nodes.

While Emcee and Chat serve as examples for illustrating the power of our architecture, they have also helped in identifying several limitations. We discuss the issues raised by our data model in Section 8.1, followed by event processing in Section 8.2, and leases in Section 8.3. We then discuss user interfaces in Section 8.4 and the interaction between *one.world* and the outside world in Section 8.5.

8.1 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 3. 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`. 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 declarative data model, which expresses schemas as data and not

as code. With a declarative 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 *declarative data models provide better interoperability than programmatic data models.*

We believe that defining an appropriate declarative data model is the single most important topic for future research into pervasive computing. The challenge is to define a data model that meets conflicting requirements. On one side, 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 Internet services and used, for example, by Microsoft's .NET platform [Thai and Lam 2001]. 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 declarative 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`.

8.2 Event Processing

Event-based programs, unlike thread-based programs, cannot store relevant state on the stack. This raises the question of how to maintain the state associated with event exchanges. In our experience, two techniques have proven effective. First, we rely on state objects in event closures to establish relevant context. For example, Emcee already needs to maintain an internal table of user records, listing, among other things, a user's name and root environment. Emcee includes this user record as the closure for any request sent to an operation. The code processing responses can then determine the appropriate context based on the closure returned with the response. Second, when performing several related operations, we rely on a worklist that is maintained by the event handler receiving responses, e.g., the `continuation` event handler in Figure 6. Upon receiving a response, the continuation removes the next item from the worklist and initiates the corresponding operation if the worklist has more items, or it invokes the appropriate logic if the worklist is empty. For instance, after activation, restoration, or migration, Chat uses such a worklist, with each item on the worklist describing a channel, to export an event handler to discovery for every subscribed channel.

Several event handlers in our applications 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. This suggests the need for better programming language support to structure event handlers. Alternatives include dynamic dispatch as provided by MultiJava [Clifton et al.

2000] or pattern matching as provided by Standard ML [Milner et al. 1997].

While we still believe that asynchronous events are an appropriate abstraction for pervasive computing, our experience with event-based programming also 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 described in Section 5.3 and measured in Section 7 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.

8.3 Leases

In our architecture, all resource access is leased, whether the resource is local or remote. 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 class 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 node is overloaded, lease renewal events may not be delivered in time. Furthermore, when a node, such as a laptop, 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.

8.4 User Interface

Emcee and Chat 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. Swing's event dispatching thread, which executes an application's user interface code, is only used while 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. Consequently, we believe that an important topic for future research into pervasive computing is how to implement scalable user interfaces. One possible approach, which is suggested by UIML [Abrams 2000], 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.

8.5 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 node'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, such as 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, especially if the outside applications are not written in Java. Because of our programmatic data model, an outside application would have to re-implement large parts of Java's object serialization, which is unnecessarily complex. We believe that moving to a declarative data model, as discussed in Section 8.1, and using a standardized communications protocol will help in providing better interoperability between pervasive applications, even if they run on different system architectures.

9. RELATED WORK

one.world relies on 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 targeted at the pervasive computing space. Furthermore, our environment abstraction is unique in that it combines persistent storage and the management of computations into a single hierarchical structure. In this section, we highlight relevant systems and discuss their differences when compared to our architecture.

Starting with Linda [Carriero and Gelernter 1986], tuple spaces have been used to enable coordination between loosely coupled services [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 coordinate between a local and a remote tuple space. Our architecture's use of tuples differs from these systems in that, as discussed in Section 4.1, our structured I/O interface is separate from remote communications and more closely resembles a database interface than Linda's *in*, *out*, and *rd* operations.

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 and functionality in *one.world*.

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]. Out 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 re-implement 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, *one.world* automates event passing through the use of wrapped 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.

Sun's Jini [Arnold et al. 1999] nominally provides many of the same services as our architecture. However, Jini embodies a different approach to building distributed applications: it builds on existing technologies, such as RPC, and is strongly object-oriented. Consequently, Jini's services are limited when compared to the corresponding services in *one.world*. For instance, Jini includes support for remote events but synchronously sends them through Java's RMI. Applications that require asynchrony thus need to implement their own event handling machinery. Furthermore, in Jini's discovery system, the service objects double as their

own service descriptors. As a result, Jini's discovery only supports early binding and simple equality queries. By comparison, SDS [Czerwinski et al. 1999] and INS [Adjie-Winoto et al. 1999] support richer service descriptions and more flexible queries. Our architecture's discovery service differs from all three systems in that it does not rely on a dedicated infrastructure; rather, the current discovery server is automatically elected from the nodes running *one.world*.

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 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 heavyweight 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.

Several other projects are exploring aspects of systems support for pervasive computing. Notably, InConcert, the architectural component of Microsoft's EasyLiving project [Brumitt et al. 2000], provides service composition in a dynamic environment by using location-independent addressing and asynchronous event passing. The Paths system [Kiciman and Fox 2000] allows diverse services to communicate in ad-hoc networks by dynamically instantiating mediators to bridge between the services' data formats and protocols.

10. CONCLUSIONS

In this paper, we have identified three principles for structuring systems support for pervasive computing environments. First, systems need to expose change, so that applications can implement their own strategies for handling changes. Second, systems need to make it easy to compose applications and services dynamically, so that they can be extended at runtime. Third, systems need to separate data and

functionality, so that they can be managed separately and evolve independently.

We have introduced *one.world*, a system architecture for pervasive computing, that adheres to these principles. Our architecture uses event-based notification to expose change. It uses nested environments as well as discovery to dynamically compose applications. It cleanly separates data and functionality: tuples represent data and components implement functionality. Additionally, our architecture provides a set of powerful services, namely operations, checkpointing, migration, and remote messaging, that serve as building blocks for pervasive applications.

Our evaluation of *one.world* shows that our architecture is a viable platform for building adaptable applications. Programming for change is in fact tractable and does not decrease programmer productivity when compared to more conventional programming styles. The performance of our implementation is sufficient for streaming audio between several users, and applications react swiftly to change, resulting in very short service interruptions if any. Based on our experience, we have identified important lessons that are applicable beyond our work. Notably, nesting is a powerful paradigm for controlling and composing applications, but asynchronous events are as hard to program as threads. We have also suggested areas for future work on pervasive computing, specifically declarative data models and scalable user interfaces. More information on our architecture, including a source release, is available at `http://one.cs.washington.edu`.

ACKNOWLEDGMENTS

Ben Hendrickson implemented parts of structured I/O, Kaustubh Deshmukh implemented a debugger, and Daniel Cheah implemented a web server running within *one.world*. We thank the students of University of Washington's CSE 490dp for serving as test subjects and Vibha Sazawal and David Notkin for their advice and assistance in evaluating the students' projects. Mike Swift provided valuable feedback on earlier versions of this paper.

REFERENCES

Abrams, M. 2000. User interface markup language (UIML). Draft specification, Harmonia, Inc., Blacksburg, Virginia. Jan. Available at `http://www.uiml.org/docs/uiml20`.

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, South Carolina, 186–201.

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

Arnold, K., O'Sullivan, B., Scheifler, R. W., Waldo, J., and Wollrath, A. 1999. *The Jini Specification*. Addison-Wesley.

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, Colorado, 267–284.

Biron, P. V. and Malhotra, A. 2001. XML schema part 2: Datatypes. W3C recommendation, World Wide Web Consortium, Cambridge, Massachusetts. May.

Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D. 1982. Grapevine: An exercise in distributed computing. *Communications of the ACM 25,* 4 (Apr.), 260–274.

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, Cambridge, Massachusetts. May.

Bray, T., Paoli, J., and Sperberg-McQueen, C. M. 1998. Extensible markup language (XML) 1.0. W3C recommendation, World Wide Web Consortium, Cambridge, Massachusetts. Feb.

Brinch Hansen, P. 1970. The nucleus of a multiprogramming system. *Communications of the 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.

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 Transactions on Computer Systems 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, California, 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, Louisiana, 44–49.

Clifton, C., Leavens, G. T., Chambers, C., and Millstein, T. 2000. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications '00*. Minneapolis, Minnesota, 130–145.

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, Washington, 24–35.

Davies, N., Friday, A., Wade, S. P., and Blair, G. S. 1998. L^{2}imbo: A distributed systems platform for mobile computing. *Mobile Networks and Applications 3,* 2 (Aug.), 143–156.

Dertouzos, M. L. 1999. The future of computing. *Scientific American 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 Jr., J. O. 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, Colorado, 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, Washington, 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, Carnegie Mellon University, Software Engineering Institute. Nov.

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, Washington, 137–151.

Foster, I. and Kesselman, C. 1997. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing 11,* 2, 115–128.

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.

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, Arizona, 202–210.

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, California, 319–332.

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, University of Washington. June.

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, Massachusetts, 93–104.

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, North Carolina, 80–93.

Jul, E., Levy, H., Hutchinson, N., and Black, A. 1988. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 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.

Kiciman, E. and Fox, A. 2000. Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. In *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing*. Lecture Notes in Computer Science, vol. 1927. Springer-Verlag, Bristol, England.

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, Cambridge, Massachusetts. Nov.

Leach, P. J. and Salz, R. 1998. UUIDs and GUIDs. Internet Draft draft-leach-uuids-guids-01.txt, Internet Engineering Task Force. Feb.

Levy, E. and Silberschatz, A. 1990. Distributed file systems: Concepts and examples. *ACM Computing Surveys 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, New York, 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, Texas, 153–164.

Martin, B. and Jano, B. 1999. WAP binary XML content format. W3C note, World Wide Web Consortium, Cambridge, Massachusetts. June.

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.

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, Arizona, 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.

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, North Carolina, 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, California, 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, California.

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, California, 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, Washington, 201–212.

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

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, New Hampshire, 110–119.

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

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.

Shapiro, M. 1986. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th IEEE International Conference on Distributed Computing Systems*. Boston, Massachusetts, 198–204.

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, Colorado, 68–77.

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

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, Louisiana, 117–130.

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, Texas, 273–284.

Thai, T. and Lam, H. 2001. *.NET Framework Essentials*. O'Reilly.

Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. 2001. XML schema part 1: Structures. W3C recommendation, World Wide Web Consortium, Cambridge, Massachusetts. May.

Thorn, T. 1997. Programming languages for mobile code. *ACM Computing Surveys 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 Concurrency 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. *Scientific American 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 Systems Journal 37,* 3, 454–474.