Programming for Pervasive Computing Environments

Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson,
Steven Gribble, Tom Anderson, Brian Bershad, Gaetano Borriello, David Wetherall

University of Washington
one@cs.washington.edu

Technical Report UW-CSE-01-06-01

Abstract

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 users 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, migration, and replication, that help to structure 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 an evaluation, which includes two case studies.

1 Introduction

In this paper, we explore how to build applications for pervasive computing environments. Pervasive computing [17, 47] 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 network services. Furthermore, many devices will be mobile and 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 transparently 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 [15]. 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 smaller and less dynamic computing environments and lack sufficient facilities to manage constant change. As a result, developers of pervasive applications have to expend considerable effort towards building necessary systems infrastructure instead of focusing on the 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 computer 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 the actual application logic and on making their applications adaptable. We have validated our approach by setting up competing teams of developers building the same applications, with one team using Java-based technologies and the other using *one.world*.

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. Section 4 describes *one.world*'s design and implementation in detail. In Section 5 we present an evaluation of our architecture, including the results of our validation case study. Section 6 discusses future work and Section 7 reviews related work. Finally, Section 8 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 or remote procedure call (RPC) packages, 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 comes at a cost in service availability and failure resilience, because it encourages a programming style in which the unavailability of a resource or a failure is viewed as an extreme case. But in an environment where tens of thousands of devices and services come and go, the unavailability of some resource is a frequent occurrence.

Second, RPC packages and distributed object systems 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 invoke each other through their interfaces. 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 is unwieldy for interfaces with numerous or complex methods. Furthermore, extensions are limited by the degree to which extensibility has been designed into the application's interfaces.

Third, distributed object systems, such as Legion [32] or Globe [44], 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. 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 code and data layout change more frequently than an object's interface, an assumption that may be less valid for a global distributed computing environment. Increasingly, common 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.

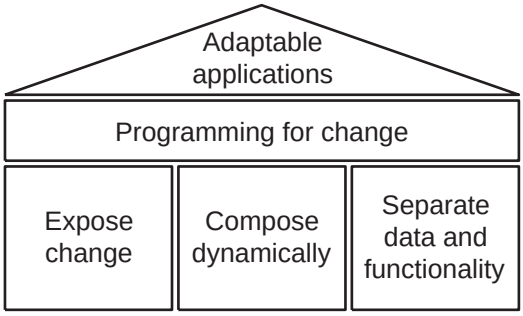

Figure 1: Overview of our approach. The three principles guide the design of our system architecture and make it feasible for application developers to program for change, resulting in adaptable applications.

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 add new services, such as caching [10, 41], content transformation [20], and content distribution [26].

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 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 [42] (beyond Java applets and JavaScript for enlivening pages) and service discovery [1, 3, 14].

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 this end, 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 that applications can implement their own strategies for handling changes. Leases [22] are an example of a suitable mechanism: they make time visible throughout a system and thus cleanly expose change. At the same time, systems need to provide primitives that simplify the task of adequately reacting to change. Examples for such primitives include "checkpoint" and "restore" to simplify failure recovery, "move to a remote node" to follow a user as she moves through the physical world, and "find matching resource" to discover suitable services on the network.

Principle 2 *Compose dynamically.*

Systems should make it easy to compose and extend applications and services at runtime. In particular, interposition on a component's interactions with other components as well as 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. This 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 an application and its data. Systems thus need to also include the ability to group data and functionality but must make them accessible independently.

Common to all three principles is the realization that systems cannot automatically decide how to react to change, because there are too many alternatives. At the same time, 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 implementation. 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.

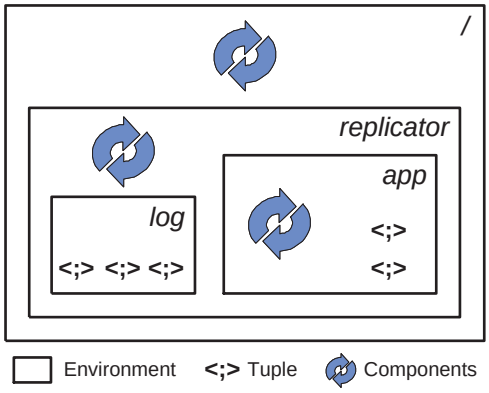

Figure 2: An example environment hierarchy. The root environment "/" hosts *one.world*'s kernel and has one child, named "replicator", which also contains active components. The *replicator* environment in turn has two children, named "log" and "app". The *log* environment only stores tuples, while the *app* environment also contains active components.

3 Architecture

In our architecture, each device typically runs a single instance of *one.world*. Each such *node* is independent of other nodes and 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 for functionality. Applications store and communicate data in the form of *tuples* and are composed from *components*. Tuples 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 importing and exporting event handlers. They statically declare which event handlers they import and export but are dynamically linked and unlinked.

Environments provide structure and control. They serve as containers for tuples, components, and other

environments. Each application has at least one environment, in which it stores tuples and in which its components are instantiated. At the same time, applications are not limited to a single environment and may span several, nested environments. Each node's root environment hosts *one.world*'s kernel. 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 [6, 18, 43] in more traditional operating systems. Figure 2 shows an example environment hierarchy.

Access to both local and remote resources is controlled by *leases* [22]. Leases limit the time applications can access resources, such as an environment's tuple storage or a communication channel, and force applications to periodically renew their interest in the resources. As a result, leases make time visible throughout the system and cleanly expose change to applications.

3.2 Services

In pervasive computing environments, location has a profound effect. As a user moves through the physical world, her applications need to be continuously available. She may carry them with her on a personal device, or they may follow her from shared device to shared device. At any location, applications need to be able to discover local resources, such as a wall display or a printer, and interact with network services. Applications also need to be prepared to operate in more limited environments. They need to provide access to shared data, even if the current location does not allow network access. Furthermore, they may have to gracefully resume after a failure, such as a user's only device's batteries running out.

Since these requirements are shared between pervasive applications, *one.world* provides a set of services that serve as common building blocks and directly help developers in making their applications adaptable. *Migration* provides the ability to move or copy an environment and its contents to another node. It affects an entire application, because both components and stored tuples are moved or copied. *Remote event passing* (REP) provides the ability to send events to remote receivers, including receivers located by service discovery. *Replication* makes stored tuples accessible on several nodes at the same time, even if the nodes are not currently connected. Finally, *checkpointing* captures the execution state of an environment tree and saves it as a tuple, thus making it possible to later revert the environment tree's execution state.

Operation	Argument	Explanation
put	Tuple	Write the tuple.
read	Query	Read a single tuple.
listen	Query	Read several tuples as they are written.
query	Query	Query for several tuples from storage.
delete	ID	Delete a tuple from storage.

Table 1: 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.

4 Design and Implementation

In this section, we describe the design and implementation of *one.world* in detail. We first present our architecture's facilities for data management in Section 4.1, followed by events and components in Section 4.2, and the environment hierarchy in Section 4.3. We conclude this section with a discussion of the current status of our Java-based implementation in Section 4.4.

4.1 Data Management

Data management in *one.world* is based on tuples. Tuples define a common data model, including the type system, for all applications and thus make it easy to store and exchange data. They are self-describing, mutable records with named and optionally typed fields and can be nested within each other. All tuples have an ID field specifying a globally unique identifier [31] (GUID) to support symbolic references and a metadata field to support application-specific annotations.

Structured I/O lets applications store tuples in environments and communicate tuples across the network on top of UDP and TCP. It exposes a common interface to storage and communications, which is summarized in Table 1. Operations are atomic so that their effects are predictable. Operations on storage can optionally use transactions to group several operations into one atomic unit. Queries for *read*, *listen*, and *query* operations support comparisons with the value of a field, including the fields of nested tuples, comparisons with the declared or actual type of a tuple or field, and negations, disjunctions, and conjunctions. To use structured I/O, applications *bind* to tuple storage or a communications end-point and then perform operations on the bound resource. All bindings are controlled by leases.

We chose tuples instead of byte strings for I/O because tuples preserve the structure of data. Tuples obviate the need for explicit marshalling and unmarshalling

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 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 [30], are relatively complex.

We chose an I/O mechanism that distinguishes between storage and communications instead of a unified tuple space abstraction [9, 21, 49] because such a mechanism better reflects how applications store and communicate data. In particular, applications often 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 support the addition of new tuples, but existing tuples cannot be changed. Furthermore, exchanging tuples through a tuple space imposes too high a performance overhead for some applications, such as streaming audio and video. Communications through storage also provide a semantic mismatch for these applications, because data should be delivered right away and not be retained in storage.

While structured I/O only provides access to local tuple storage, replication makes tuples accessible to applications on multiple nodes, even if the nodes are disconnected. *one.world*'s replication layer is patterned after Gray et al.'s two-tier replication model [23]. A master node owns all data and replicas have copies of that data. Replicas can either be connected or disconnected. In connected mode, updates are final and performed directly on the master. In disconnected mode, updates are tentative and logged on the replica. When a replica becomes connected again, it synchronizes with the master by replaying its log against the master and by receiving updates from the master. The replica may then disconnect again or continue in connected mode.

We chose two-tier replication over Bayou's epidemic replication model [37, 40] for two reasons. First, two-tier replication is easier to explain to users. Tentative updates may only change once, during synchronization, and not repeatedly. Second, two-tier replication avoids system delusion [23]. Delusion occurs when large numbers of replicas reconcile with each other repeatedly in the absence of a master and consequently diverge further and further from each other.

The implementation of replication is not part of *one.world*'s kernel and makes extensive use of its core primitives. A replica is structured as shown in Figure 2 (with the log environment also containing active components). The replicator interposes on the application's access to tuple storage. It logs updates in the log environment when in disconnected mode and forwards them to the master using REP when in connected mode. On reconnection of a disconnected node, the log is sent to the master by migrating a copy of the log environment. Similarly, updates are sent from the master to the replica by migrating an environment containing such updates.

Our implementation has two important additional features over two-tier replication as described in [23]. First, it allows for continuous operation during synchronization, thus improving usability. Second, the master and its replicas can migrate to different nodes without disrupting operation. Migrating the master is useful when, for example, upgrading the computer the master is running on; migrating a replica is useful for applications that follow a user as she moves through the physical world.

4.2 Events and Components

Control flow in *one.world* is expressed through asynchronous events that are processed by event handlers. Events are simply 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 exceptional conditions during event delivery and processing as well as the response for request/response interactions. Furthermore, all events have a closure field. 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.

Application functionality is implemented by components, which import and export asynchronous event handlers. Components are instantiated within specific environments and, in their constructors, declare which event handlers they import and export. They can be linked and unlinked at any time. 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 and relink and unlink components as needed.

To implement asynchronous event handling, each environment provides a queue of pending $\langle$*event handler, event*$\rangle$ invocations as well as a pool of one or more threads to perform such invocations. When sending an event between components in different environments, the corresponding event handler invocation is automatically enqueued in the $\langle$*event handler, event*$\rangle$

queue of the target environment. When sending an event between components in the same environment, the event handler invocation is implemented as a direct method call. 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.

Remote event passing (REP) provides the ability to send events to remote receivers. It supports both point-to-point communications and service discovery, including early and late binding [1], as well as anycast and multicast, through only three simple operations: *export*, *send*, and *resolve*. The *export* operation makes an event handler accessible from remote nodes through a symbolic descriptor. The resulting binding between the event handler and descriptor is leased. Furthermore, when exporting an event handler for service discovery, the binding is propagated to the discovery server for the local network. The *send* operation sends an event to previously exported event handlers by using an exported descriptor or discovery query as the remote address. When using late binding, the event is routed through the discovery server, where the discovery query is resolved. Otherwise, the event is sent directly to the node exporting the targeted event handler. Finally, the *resolve* operation looks up event handlers on the discovery server.

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 react to real world events, such as a meeting being started or a person leaving a room. Second, unlike threads, which 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, for example, event priorization or checkpointing and migration. Finally, taking a cue from other research projects [11, 24, 25, 36] 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. For instance, the uniform event handling interface enables a flexible component model, which supports the linking of any imported event handler to any exported event handler. Furthermore, it makes it possible to combine point-to-point communications and service discovery within a simple and narrow interface to REP. At the same time, the uniform event handling interface does not prevent the expression of typing constraints. When components declare the event handlers they import and export, they can optionally specify the types of events sent to imported event handlers and processed by exported event handlers.

4.3 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 and do not hide them behind a unifying interface. The environment abstraction was inspired by the ambient calculus [8]. Similar to environments, ambients serve as containers for data, functionality, and other ambients. But, while ambients are formal constructs used to reason about mobile computations, environments are actual entities in the *one.world* architecture that 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 represented as tuples and loaded from environments. Checkpointing captures the execution state of an environment tree, including application components and pending ⟨*event handler, event*⟩ 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 stored tuples, to a remote node.

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, that is, it 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 ⟨*event handler, event*⟩ 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. During serialization, all 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 structured I/O. Applications need to restore nulled out handlers themselves by relinking or rebinding after restoration or migration.

The nesting of environments is relevant for the following three features. First, environments can be isolated from each other and are subject to hierarchical resource controls for CPU slices, memory, and tuple storage, similar to those described in [5, 43]. Second, 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 she moves through the physical world can migrate *any* application, simply by dynamically embedding the application in its environment. Third, 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. Each environment imports an event handler called "main". This event handler 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, including that the environment has been activated, restored, or migrated, or that it is about to be terminated.

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 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. Furthermore, by linking to the monitor handler, an application can interpose on all events sent to a descendant's request handler. For example, the replicator in Figure 2 can intercept the application's binding request for tuple storage by linking to its monitor handler. It can then return an event handler that not only performs structured I/O operations on the application's tuple storage but also logs these operations in the log environment (see Figure 2). This use of the *request/monitor mechanism* is illustrated in Figure 3.

To enforce the nesting of environments, *one.world* limits access to tuple storage and operations on environments 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 makes interposition trivial and thus 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. Finally, the same mechanism can be used to provide security by interposing a reference monitor [2] and auditing by logging an application's request stream. Its thus obviates the need for fixing a particular security mechanism or policy in *one.world*'s kernel.

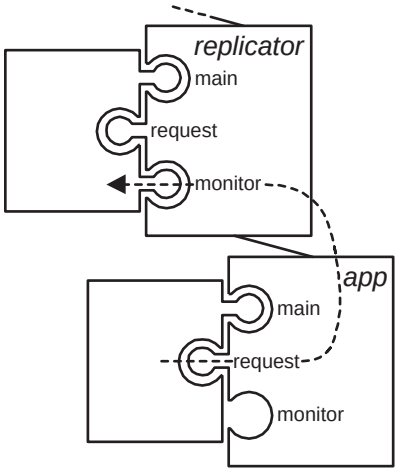

Figure 3: Illustration of the request/monitor mechanism. Boxes on the left represent application components and boxes on the right represent environments. The *app* environment is nested within the *replicator* environment. The *replicator* environment's monitor handler is linked and thus intercepts all events sent to the *app* environment's request handler. The use of the main handler is explained in the text.

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 and thus increases an application's exposure to failures, while also hiding the cause of failures from the application. Furthermore, we argue that nulling out event handlers does not place an additional burden on developers, because applications already need to be prepared to reacquire resources, for example, to rebind a structured I/O resource because the lease has expired.

4.4 Implementation

We have implemented *one.world* largely 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 [35] to implement reliable tuple storage. Our architecture currently runs on Windows and Linux PCs, and a port to Compaq's iPAQ handheld computer is under way. The implementation of *one.world* itself has approximately 16,000 non-commenting source

statements (NCSS) with an additional 5,400 NCSS for our replication layer. Our entire source tree, including regression tests, benchmarks, and applications, has approximately 44,000 NCSS. A Java archive file with the binaries for *one.world* including replication is 675 KB. The GUID generation and Berkeley DB libraries require another 485 KB on Windows systems.

The implementation currently lacks support for transactions and class loading from environments, as well as isolation and resource controls. The implementation of transactions and class loading from environments should be straightforward. The implementation of isolation and resource controls can either be done in pure Java [13, 46], albeit at some performance cost, or by modifying the Java virtual machine [4, 5]. We plan to implement these features in the near future. Furthermore, we believe that their absence does not impact the overall results presented in this paper. After all, we focus on how to build adaptable applications for pervasive computing environments and not on how to provide protection for language-based systems.

5 Evaluation

In this section, we present a thorough evaluation of *one.world*. We first characterize the basic performance of our architecture in Section 5.1. We follow with the results of a case study on students using *one.world* to build real applications in Section 5.2. We conclude with an evaluation of our own experiences with implementing replication in Section 5.3. Overall, the results show that *one.world* has acceptable performance and is a viable platform for building pervasive applications. They also suggest that our architecture requires additional support for better structuring application code.

5.1 Basic Performance

To determine the basic performance of *one.world*, we measured the performance of microbenchmarks for tuple storage, tuple communications, remote messaging, and migration. Where applicable, we compare the results with microbenchmarks for the underlying storage and communication services as well as related technologies, including IBM's T Spaces 2.1.2 [49]. All measurements were performed using off-the-shelf PCs, with Pentium III 800 MHz processors and 256 MB of RAM, running Windows 2000. The PCs are connected by a 100 Mb switched ethernet. We use Sun's HotSpot client virtual machine 1.3.0 and Sleepycat's Berkeley DB 3.2.9. Reported results represent the average of 100 benchmark runs.

Test	BDB	Serial BDB	*one.world*
Read	0.24	0.88	1.23
Write	25	25	25
Query	44	640	860

Table 2: Latency of storage operations in milliseconds. Queries retrieve all stored tuples. *BDB* shows performance of the Berkeley DB without serialization, and *Serial BDB* shows performance of the Berkeley DB with serialization.

Test	T Spaces	*one.world*
Read	2.9	1.2
Write	4.5	25
Query	18	690

Table 3: Latency of storage operations in milliseconds. Queries are by field value and match 23 tuples on average.

For tuple storage, we compare the performance of structured I/O with that of the Berkeley DB, the underlying storage used by structured I/O. The Berkeley DB benchmarks are performed both with and without serialization. We also compare the performance of structured I/O with that of T Spaces, an in-memory tuple space that uses periodic checkpointing for persistence. The microbenchmarks read, write, and query for tuples representing personal contacts (as might be used by a personal information manager). All tests use a store with 1,000 such tuples. Table 2 shows the results for the comparison between structured I/O and the Berkeley DB. Writes are dominated by the cost of forcing each write to disk and thus perform the same in all cases. The overhead for structured I/O reads and queries over the Berkeley DB with serialization is a reasonable 40% and is likely due to the cost of switching threads during event delivery. Table 3 shows the results for the comparison between structured I/O and T Spaces. Structured I/O is in the same performance class as T Spaces for reads and writes. Writes are slower for structured I/O because structured I/O forces each write to disk, while T Spaces only performs periodic checkpoints. Using similarly relaxed durability for structured I/O, write latency drops to 0.73 ms. Field queries perform considerably worse for structured I/O, as we currently only index tuple IDs.

For tuple communications, we compare the performance of structured I/O with several other communication methods. These benchmarks repeatedly send 100 bytes of data from one node to another by sending a tuple containing a byte array or by sending the raw bytes. We use 100 bytes of data because overhead is more pronounced for small payload sizes. Table 4 shows the results. Structured I/O over UDP and TCP each achieve

Transport	Throughput
DatagramIO	120
UDP (tuples)	160
UDP (bytes)	1,600
NetworkIO	180
TCP (tuples)	230
TCP (bytes)	9,100
T Spaces	24

Table 4: Networking throughput in kB/sec. *DatagramIO* is structured I/O over UDP, and *NetworkIO* is structured I/O over TCP; the corresponding rows are highlighted.

System	Latency
RMI	2.8
REP	3.6

Table 5: Remote messaging latency in milliseconds.

throughput within 30% of sending tuples directly over the underlying transport. However, this performance is well below that of sending raw bytes over UDP and TCP. The overhead is due to the increased space required for serialized tuples and the low performance of Java serialization. Structured I/O achieves at least five times the throughput of T Spaces. This comparison illustrates the effects of conflicting requirements for tuple spaces. On one side, tuple spaces are expected to provide long-term storage for tuples. On the other side, tuple spaces are also used for immediate communication. Structured I/O strikes a better balance between these requirements by separating storage and communications but by also providing a common interface.

For remote messaging, we compare the latency of a request/response interaction using REP and Java's remote method invocation (RMI). Both benchmarks send an event to a remote node, where it is immediately sent back to the sender. For RMI, the event is the only argument to the remote method and is returned as its result. The results in Table 5 show that the performance of REP is comparable to that of RMI, with REP suffering a 25% performance penalty.

For measuring the performance of migration, we use a small application that repeatedly moves itself across a set of nodes in a tight loop; its environment contains a varying number of tuples. Just as with tuple communications, we use tuples with 100 bytes of data because overhead is more pronounced for small tuples. Table 6 shows the average latency of a move from one node to another, given a loop of three nodes. This benchmark is hard to compare with other mechanisms because of the uniqueness of our move primitive. However, we believe that its performance is reasonable, given the presence of a human waiting for a migration to complete.

Tuple count	Move Latency
0	0.39
100	0.7
1,000	2.7
10,000	23

Table 6: Migration latency in seconds. *Tuple count* is the number of tuples in the migrating application's environment.

Overall, the results show that *one.world*'s primitives have reasonable performance, given the high cost of serialization. We plan to add indexing to our implementation of tuple storage to improve the performance of field queries. Furthermore, we are considering alternatives to reduce the overhead of serialization. Options include the use of pre-serialized objects [48] and dynamic code generation to specialize serialization.

5.2 Usability and Effectiveness

To evaluate *one.world*'s usability and effectiveness for building real applications, we conducted an experiment in the form of a senior-level undergraduate project course. After ten introductory lectures, the nine students in the class split into two teams that developed a music sharing system and a universal inbox. Each team split into two subteams, with one subteam using existing Java-based technologies and the other subteam using *one.world*. Since both subteams implemented the same application, this experiment lets us compare our architecture with other approaches to building distributed systems. The results presented here are based on weekly meetings with the teams, end-of-term interviews, and the teams' final presentations and reports.

The first team developed a music sharing system, which relies on a dynamically configured hierarchy of directory nodes to organize searches. The Java subteam implemented the application in plain Java, without using additional technologies. Results for the music sharing team are incomplete; students barely completed the implementations, although they did demonstrate working applications. The students' experiences suggest that our architecture's support for queries as part of structured I/O and for asynchronous messaging through REP clearly simplified the implementation on top of *one.world*. In contrast, the Java subteam implemented querying and asynchronous messaging from scratch.

The second 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 Java version uses Jini [3] for service configuration and T Spaces [49] for storing repository data.

The students' experiences support our argument from Section 2 that extending programming models for single-node applications to distributed systems makes it difficult to build adaptable applications. Jini relies on Java's RMI to access remote resources and is designed to simplify the conversion of existing code into Jini services. The Java subteam exploited this and originally implemented individual services, such as the message router or data repository, as stand-alone, single-node applications. Students subsequently "jinified" the applications and iteratively refined them as network services. While the conversion of an application into a bare-bones Jini service is simple, turning a minimal Jini service into a full-blown Jini service is an arduous process. This refinement process involved repeatedly testing the system to identify potential failure conditions and then adding code to account for such conditions. Students also had to work around the synchronous design of RMI. While Jini includes support for remote events, they are implemented as synchronous invocations through RMI and thus expose services to possibly indefinite delays, for example, because the service receiving an event is buggy and hangs. The completed implementation still reflects the difficulties of the refinement process and has relatively few services, with each of these services representing a single point of failure for all users.

In contrast, the *one.world* subteam found our architecture's primitives well matched to their needs. In particular, local tuple storage, REP's support for late binding, and the signaling of exceptional conditions through regular events simplified the implementation. As a result, the completed implementation does not require a centralized data repository and separates each user's email management into an independently running service. The biggest challenge for the *one.world* subteam was managing asynchrony: the highly structured, event-based programming model clearly exposed failures and forced students to consider appropriate failure of recovery strategies from the beginning. From a software engineering viewpoint, the students found that components required too much repetition and that event handlers handling several types of events were not very modular. They thus created several abstract components to avoid code repetition and facilities for dynamic event dispatch to make event handlers more modular. This suggests that our architecture requires better support for avoiding code repetition and for structuring event handlers, such as the dispatch facilities provided by MultiJava [12].

5.3 Replication Case Study

Since our replication layer (see Section 4.1) is not part of *one.world*'s kernel, we used the process of implementing it as a case study on how to structure applications in our architecture. The implementation makes extensive use of *one.world*'s core features and illustrates the power of a design that follows the three principles presented in Section 2:

1. *Expose change*. The consistent use of leases cleanly exposes change, such as a replica getting disconnected from the master.

2. *Compose dynamically*. The nesting of environments makes it easy to dynamically compose functionality. In particular, the request/monitor mechanism let us interpose replication on an application's tuple storage. Furthermore, we used the same mechanism for communication between migrated log environments and the master.

3. *Separate data and functionality*. The separation of data and functionality provides considerable flexibility. Since events are tuples, we directly logged structured I/O events on the replica. Furthermore, it was easy to support application-specific reconciliation, simply by adding the corresponding component to the log environment.

Additionally, migration and REP provide powerful primitives that cover the spectrum between collocation and remote interaction. On one side, we relied on migration to colocate a replica's log with the master during reconciliation; on the other side, we used REP during connected operation.

Figure 4 illustrates the performance of a replica under changing network conditions. We measure the performance of a benchmark that repeatedly writes a tuple in a tight loop, and use the same experimental set-up as in Section 5.1. The replica starts in disconnected mode and logs all updates locally. When the replica discovers the master, it copies the log to the master and reconciliation begins (1). Throughput drops while the replica is connected to the master because updates are forwarded to the master. After reconciliation has completed, the master begins copying updates destined for the replica into a separate response environment (2). After the response environment has arrived at the replica and its updates have been applied, the replica enters connected mode and starts to purge old log entries (3). When purging is complete, throughput improves slightly (4). Finally, the master becomes disconnected again (5), and the replica returns to disconnected mode (6).

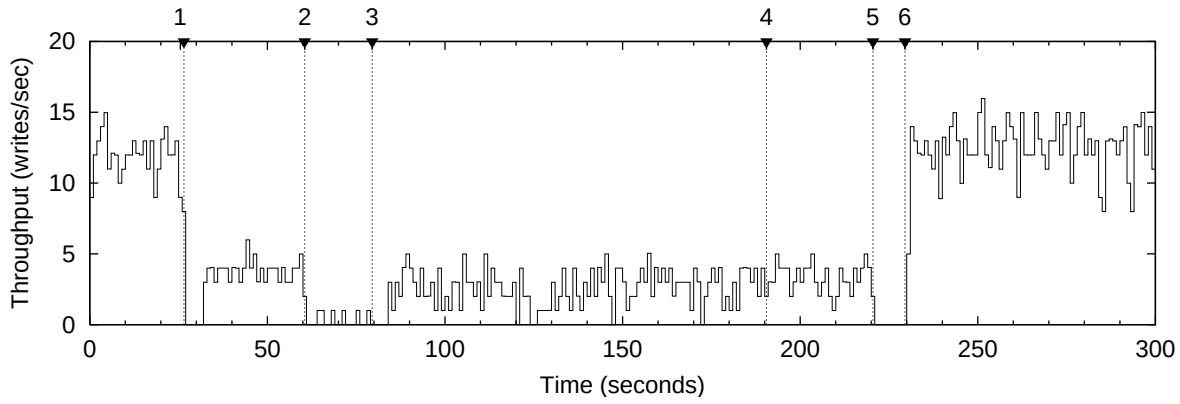

Figure 4: Performance of a replica under changing network conditions. The replica starts in disconnected mode and discovers the master at point 1. It becomes disconnected again at point 5 and re-enters disconnected mode at point 6. The other points are explained in the text. Throughput is generally limited by structured I/O forcing each write to disk.

5.3.1 The Logic/Operation Pattern

During development of *one.world*, one important concern was how to write applications in the face of considerable uncertainty. Since access to resources is leased, most operations may fail and an application must be prepared to adequately react to such failures. In our experience, established styles of event-based programming, such as state machines, are only manageable for very simple applications. The single biggest challenge in implementing replication was deciding how to appropriately react to failures and how to structure the corresponding failure recovery code. After some experimentation, we found the following approach, which we call the *logic/operation pattern*, particularly successful.

Under the logic/operation 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. Operations are interactions that may fail, such as reading a tuple or sending a remote message. The implementation of operations includes all necessary failure detection and recovery code. For example, when reading a tuple, the corresponding operation rebinds if the lease for tuple storage has expired. It also sets a timeout and retries if the timeout expires. A failure condition is reflected to the appropriate logic only if recovery fails repeatedly or the failure condition cannot be recovered from in a general way.

Event handlers for logic and operations can easily be composed with each other. Figure 5, for example, illustrates a structuring reminiscent of an if-then-else construct. Other possible structurings include gotos and loops. More importantly, the sequence of logic and operations in the figure itself constitutes an operation and can be used just like a basic operation. The

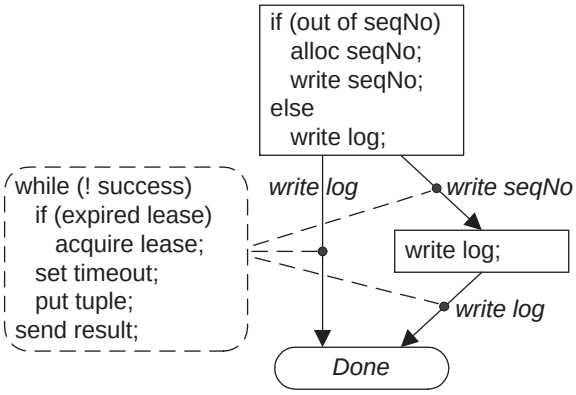

Figure 5: An example of the logic/operation pattern, which writes a log record and, if necessary, also allocates additional sequence numbers. Boxes represent logic; the pseudo-code shows how operations are invoked by logic. Arrows represent operations, which include necessary failure recovery code. The call-out illustrates the implementation of the operations.

logic/operation pattern thus approximates a function call in traditional programming languages and provides programmers with a familiar framework for building complex applications in *one.world*. At the same time, use of the logic/operation pattern does not relieve developers from deciding on appropriate failure recovery strategies and correctly implementing an application's logic.

6 Future Work

Future work on *one.world* will focus on three areas: the core architecture, support for debugging, and applications. As discussed in Section 4.4, our implementation

currently lacks support for transactions and class loading from environments, as well as isolation and resource controls. We plan to implement these features in the near future and also add support for indexing to tuple storage (see Section 5.1). More importantly, having identified the logic/operation pattern as a viable way to structure applications in our architecture, we will integrate it into the core architecture. We will also build a library of common operations that can be parameterized by failure recovery strategy.

While debugging distributed applications is difficult enough, debugging applications in *one.world* is even harder due to its asynchronous programming model and inadequate support for tracing application events. Our architecture logs unhandled events, including a stack trace, and provides a runtime option to log all kernel events. Additionally, both we and the students in our case study have made extensive use of the debug-by-printf approach to log relevant application events. However, instrumenting applications to log events is cumbersome. Furthermore, analyzing event logs is hard because, in an asynchronous system, events may be reordered and causal relationships may not be obvious from the log. Therefore, we plan to add better debugging support to *one.world*. The request/monitor mechanism lets us easily interpose a debugger on an application, and events can already be annotated with metadata, making it possible to track causal relationships even across nodes.

Finally, to gain more experience with our architecture, we plan to build additional applications. We will focus on applications that actually enhance our own productivity, for example, by better integrating our computing environments at home and at work. Furthermore, we plan to collaborate with other researchers and support them in building their own pervasive applications on top of *one.world*. To this end, we distribute source releases of our architecture and will hold workshops. One research group is already using *one.world* to implement a pervasive environment for a biology laboratory.

7 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. Leases have been used to control write access to cached files [22], access to remote resources in general [3], and even the lifetime of stored objects [21]. Starting with Linda, tuple spaces have been used to enable coordination between loosely coupled services [9, 21, 49]. The Information Bus provides similar functionality based on a publish/subscribe paradigm [34]. While it nominally uses objects, exchanged information is represented by self-describing data objects, which are similar to tuples in *one.world*. Asynchronous events have been used across a wide spectrum of systems, including networked sensors [25], embedded systems [11], user interfaces [38], and large-scale servers [24, 36]. Finally, several projects have investigated the use of service discovery to simplify the configuration of distributed systems [1, 3, 14].

A significant number of projects have explored migration in distributed systems [33]. Notable examples include migration at the operating system level, as provided by Sprite [16], and at the programming language level, as provided by Emerald [27, 39]. In these systems, providing support for a uniform execution environment across all nodes and migration of application and execution state has resulted in considerable complexity. In contrast, many mobile agent systems, such as IBM's aglets [29], 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. 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 persistent data. Furthermore, its use of asynchronous events greatly simplifies the migration of execution state.

Several efforts, including Globe [44], Globus [19], and Legion [32], are exploring 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, by extending models for distributed computing originally developed for small and relatively static computer networks, 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 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 [7], provides service composition in a dynamic environment by using location-independent addressing and asynchronous event passing. The Paths system [28] allows diverse services to communicate in ad-hoc networks by dynamically instantiating mediators to bridge between the services' data formats and protocols.

8 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 so that they can evolve independently.

We have introduced a system architecture for pervasive computing, called *one.world*, that adheres to these principles. Our architecture uses leases to expose change. It uses nested environments as well as late binding 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 migration, remote messaging, replication, and checkpointing, 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. More information on our architecture, including a source release, is available at `http://one.cs.washington.edu`.

Acknowledgments

We thank Ben Hendrickson for implementing parts of the storage subsystem. We thank Vibha Sazawal and David Notkin for their advise and assistance in our comparative evaluation of *one.world* and the students of University of Washington's CSE 490dp course for serving as test subjects. We thank Mike Swift for his valuable comments on an earlier version of this report.

References

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

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

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

[4] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 333–346, Oct. 2000.

[5] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the design of Java operating systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 197–210, June 2000.

[6] P. Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 250, Apr. 1970.

[7] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. EasyLiving: Technologies for intelligent environments. In *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing*, pages 12–29, Sept. 2000.

[8] L. Cardelli. Abstractions for mobile computations. In Vitek and Jensen [45], pages 51–94.

[9] N. Carriero and D. Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[10] A. Chankhunthod, P. B. Danzig, C. Needaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 153–163, Jan. 1996.

[11] P. Chou, R. Ortega, K. Hines, K. Partridge, and G. Borriello. ipChinook: An integrated IP-based design framework for distributed embedded systems. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 44–49, June 1999.

[12] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. 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*, pages 130–145, Oct. 2000.

[13] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications '98*, pages 21–35, Oct. 1998.

[14] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 24–35, Aug. 1999.

[15] M. L. Dertouzos. The future of computing. *Scientific American*, 281(2):52–55, Aug. 1999.

[16] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice and Experience*, 21(8):757–785, Aug. 1991.

[17] M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next century challenges: Data-centric networking for invisible computing. In *Proceedings of the 5th ACM/IEEE*

International Conference on Mobile Computing and Networking, pages 256–262, Aug. 1999.

[18] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 137–151, Oct. 1996.

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

[20] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 78–91, Oct. 1997.

[21] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[22] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, Dec. 1989.

[23] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, June 1996.

[24] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 319–332, Oct. 2000.

[25] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Nov. 2000.

[26] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek. The measured performance of content distribution networks. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, May 2000. `http://www.terena.nl/conf/wcw/Proceedings/S4/S4-1.pdf`.

[27] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb. 1988.

[28] E. Kiciman and A. Fox. Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. In *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing*, Sept. 2000.

[29] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.

[30] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document object model (DOM) level 2 core specification. W3C recommendation, World Wide Web Consortium, Nov. 2000.

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

[32] M. Lewis and A. Grimshaw. The core Legion object model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 551–561, Aug. 1996.

[33] D. Milojičić, F. Douglis, and R. Wheeler, editors. *Mobility—Processes, Computers, and Agents*. ACM Press. Addison-Wesley, Feb. 1999.

[34] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus — an architecture for extensible distributed systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 58–68, Dec. 1993.

[35] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track, 1999 USENIX Annual Technical Conference*, pages 183–192, June 1999.

[36] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 199–212, June 1999.

[37] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 288–301, Oct. 1997.

[38] C. Petzold. *Programming Windows*. Microsoft Press, 5th edition, Nov. 1998.

[39] B. Steensgaard and E. Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 68–77, Dec. 1995.

[40] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 172–183, Dec. 1995.

[41] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design considerations for distributed caching on the Internet. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 273–284, June 1999.

[42] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, Sept. 1997.

[43] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Proceedings of the 8th ACM SIGOPS European Workshop*, pages 111–117, Sept. 1998.

[44] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, 1999.

[45] J. Vitek and C. D. Jensen, editors. *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[46] T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, D. Hu, and D. Spoonhower. J-Kernel: a capability-based operating system for Java. In Vitek and Jensen [45], pages 369–393.

[47] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, Sept. 1991.

[48] M. Welsh and D. Culler. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, June 2000.

[49] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.