

# The Design and Evaluation of a Storage System for Pervasive Computing

Eric Lemar

May 18, 2001

## Abstract

Many interesting pervasive applications require the maintenance of state, but few pervasive computing systems provide a compelling system for managing such state. Although conventional storage abstractions can fill some of these needs, the unique character of the pervasive environment suggests slightly different answers. Pervasive applications often have to work with data in forms they were not directly coded to support, need to be able to extend the behavior of existing applications, and must support mobility across a wide range of devices. The *one.world* storage architecture supports the needs of pervasive computing and overcomes many of these limitations in other systems. This paper examines the design decisions made in the *one.world* storage architecture and API and provides an initial look at the lessons learned from the implementation of this design.

## 1 Introduction

The pervasive computing model [13] assumes that in the near future computing power will pervade our everyday environment. Although computers will be ubiquitous, the specific equipment available to complete our tasks will change rapidly due to the appearance or removal of computing devices and due to the movement of the user from one device to another. Network connectivity is an important feature of this environment, but this connectivity is expected to be neither constant nor always to a single, universal, Internet due in part to physical constraints such as power for mobile devices. The ubiquity of computational power combined with task oriented usage patterns will foster a loosely coupled system in which applications must coexist without explicit effort to make them coexist. The behavior of applications must be expandable to meet the changing needs of their users: a single application developer will not be able to explicitly code for all of the behavior needed by every individual in such a dynamic world. In short, applications must be able to deal with all sorts of change, including an evolving and varying computing environment, an evolving pattern of use, and evolving data formats. The focus of the *one.world* project is to provide a software platform and application programming model that allows for this sort of use and which eases the creation of applications which take advantage of this new computing infrastructure.

Many of the most interesting pervasive applications are very data centric. In particular, some of the early adopter applications on devices such as Palm Pilots have been contact databases, scheduling, real-time bus schedule delivery, news updates, and similar applications where the application is essentially a means of managing and displaying a moderate amount of data. In addition to applications designed around the management and display of moderate amounts of known data, we expect services and applications that manage large amounts of unknown data types to be important. Applications of this sort include conventional data management applications that replicate arbitrary data or sift through incoming data according to user specified filters. It seems likely that in a future flexible computing environment it will be impossible for an application builder to envision all of the necessary policies for moving and filtering data and an ability to add such abilities as a separate user specific component will be essential. For instance, the programmer of a data editor will be unable to foresee all important policies for moving this editor from computer to computer following a laboratory worker around his laboratory or a college student around his campus. Likewise, a single application will not be able to foresee all of the important filtering or searching policies and data types that a given user will require. Motivated by the storage needs of pervasive applications, *one.world* provides

a tightly integrated storage system based on the idea of flexibility and expandability. This paper describes and justifies the design of this storage system. The current implementation is briefly outlined, is not the focus of this paper. Finally, an initial evaluation of experiences with the existing implementation and API will be presented.

## 2 Background

We first provide a quick overview of *one.world* to aid in the understanding the storage design. In previous papers we have discussed the guiding principles of *one.world*[5], but in short we believe that the focus of a pervasive system such as *one.world* is to allow all manner of change to be easily introduced by and reported to applications. As an example of a specific design rule motivated by this principle, we believe that application functionality and data should be kept separate in order to allow these aspects of the system to evolve independently. As a second design rule, we believe that applications should be given the tools to cope with change in their physical environment. In particular, changes in the environment of the application (for instance moving the application from one machine to another) must not be hidden from the application. Unlike conventional systems, such as Sprite[2], which attempt to hide such changes, we feel that applications must be informed and be prepared to deal with them by explicitly rebinding to resources since the correct behavior in such situations cannot be determined at the system level. *one.world* provides a uniform runtime environment with a small guaranteed core of available services which are supported by all machines that are part of the pervasive computing environment.

In its part in achieving these goals, we feel that the store must accomplish several goals. It must provide a separation of data from functionality to promote separate evolution as well as easy application independent access of data. It must provide mechanisms for uniformly reporting change to the application. It must provide well defined semantics in the case of concurrent access to support sharing of data. Additionally, the API of the storage system must be designed to allow the system to be easily extended outside of the kernel to add features such as data replication or flexible access control to existing storage clients. Finally, in order to reduce the total number of abstractions we hope to provide an an API that can be shared with inter-process data communication. We note that such an arrangement has worked well in systems such as Unix through the of a common abstraction, the file descriptor, and we wish to replicate this success.

*one.world* uses a form of semi-structured data, the *tuple*, as its basic data model. This model is inspired both by conventional tuple systems [4] [14] which were designed to support communication between loosely coupled systems as well as XML, which was largely designed to support a more data-driven exchange of information between loosely coupled systems[1]. Tuples contain named and optionally typed fields holding data. Tuples fields can contain sub-tuples, allowing a hierarchical naming structure. A tuple model was chosen instead of a bare XML model for several reasons. First, since tuples are heavily used programmatically (as mentioned later, they form the basis of all inter-component control flow), we need a mapping other than a straight textual representation, making a direct use of XML impractical. Second, again to make access straightforward, we want at most a subset of the XML functionality: in particular, several constructs such as the repeated use of the same element tag are inconvenient to access programmatically. Finally, we desire a typed system, and XML offers no innate typing. Although it is possible to map our data model into XML, it seemed easiest and most efficient to simply implement this model directly and to use XML for what it is good at: talking to unknown external systems. Although not currently implemented, we plan to support a mapping to and from XML in order to communicate with non *one.world* systems. Unlike tuples in a conventional tuple system, our tuples also contain a required Id field containing a Guid [7] which can be used as a key to an individual tuple. The exact semantic meaning of the Id field is user-defined and simply provides a common field that services can use to reference tuples.

*one.world* is currently implemented in Java and uses Java bytecode as its universal binary format. For several reasons including resilience under load, loose coupling, and ease of state inspection, the *one.world* execution system is a fully asynchronous event based system. All execution outside the kernel, and most within the kernel, happens as a result of processing of asynchronous events. Events are simply tuples with a field indicating a source event handler used for addressing responses or errors, and a closure which is echoed by responses to allow easy response demultiplexing. We have a component model, with inter-component invocation accomplished by sending an event to an EventHandler provided by another component. An

EventHandler is simply Java object implementing an interface with a single function void handle(Event) taking an Event object.

The contract with the system is that any processing for an event must be of finite duration. Events are placed on event queues which are then serviced by pools of one or more threads that pull events off the queues and execute them. Event delivery semantics are “best effort, at most once” delivery, with no ordering guarantees. All user-space operations are non-blocking. Most operations are expected to be request/response style interactions. The failure model is also somewhat different than is normally presented to an application. In a typical request/response pattern, there are three possible outcomes: 1) both the request and the response are handled as expected, 2) the request event is dropped and thus no action is taken and no response is generated, or 3) the request event is delivered and action is taken, but the response event is dropped. A dropped local event will generally be signalled to the application through an asynchronous exceptional event, but in extreme cases this loss notification may be silently dropped. For local machine communication, the most common reason for a dropped event is a full event queue. Applications are responsible for detecting and reacting to any dropped events using application dependent logic (in many cases this will be through the use of timeouts and retries). This asynchrony and error model requires somewhat different API's than in a conventional system.

All resources in one.world are leased. A leased resources consists of two things: a pointer to the resource, generally an event handler, and a lease to use this resources. This lease contains the duration for which the lease is valid as well as a second event handler which is used for managing the lease. Leases are limited to a finite duration, but can generally be renewed indefinitely (a service may choose not to allow lease renewal). Note that although a lease generally confers something of a guarantee of accessibility, a service can explicitly cancel a lease if some exceptional condition, such as the deletion of the resource, requires this.

Stepping up a level, the basic *one.world* system provides data storage (the system being described), remote tuple communication, remote event passing, and resource discovery. There is built in support for checkpointing and restoring executing code as well as for moving running applications between machines. *one.world* provides a timer producing either one-shot or periodic timer firings which can be used to control the execution of non-continuously computing applications.

### 3 Current storage solutions

Before exploring the *one.world* storage system, it is useful to look at the suitability of existing storage systems in the pervasive arena. In particular, there are several properties of importance to pervasive computing useful for evaluating these alternatives. *First*, we want a system with a good method of grouping data so that related data can be managed as a unit for operations such as resource control, movement, or replication. *Second*, we want a system that provides flexibility in the type of data we can store and access. In particular, we want a system flexible enough to efficiently store and access arbitrary tuple types. *Third*, we want a storage system which provides safe and well-defined concurrent access to data. We expect that many important data sources, such as a list of PIM contacts, will be accessed by many independent programs and we need a safe way to support such access. *Finally*, we want a storage model that provides useful operations for pervasive style uses. This includes, but is not limited to, a good way to query arbitrary data and a good way to move data from node to node in the pervasive environment.

In addition to the desired features of the system, there are a couple of more stylistic requirements for our storage system. The API must be designed in a manner compatible with the “best effort, at most once” asynchronous event passing model for invocation. Additionally, although the flexibility and power of the system are important, the simplicity of the system as measured both by the simplicity of the interface and by the simplicity of semantic interpretation of operations is crucial. The advantages of simplicity are that the user can learn the interface more easily and, much more importantly, that a simple interface takes much less work to intercept, understand, and extend. For a system which envisions dynamic composition and extension to be a common need, this element is an important design consideration.

Four types of existing systems that will be discussed:

- *File systems and distributed file systems.* File systems are the dominant storage system for conventional desktop workstations. Distributed file systems have become one of the dominant ways of sharing data

between conceptually equivalent computers (for instance, in a corporate network) and have also been used for some more advanced systems such as the Sprite system, which supported process mobility among a set of machines. One nice property of the filesystem approach to data storage is that these systems present a structured hierarchical organization of storage. Filesystems also have the advantage that they provide a very low level abstraction which allows arbitrary data to be stored. Unfortunately, the filesystem generally maintains only a small, fixed amount of metadata (file name, creation date, etc) with the typing and delineation of internal structure being application dependent. Although this has the advantage of being very flexible, this makes programatic access of unknown data types, which we consider a crucial capability for pervasive computing, challenging or impossible. The lack of structuring in most file systems also makes it difficult to embellish existing formats. Some systems, such as NeXT and even NTFS under windows NT, have attempted to work around some of these issues through the use of multiple named streams, but these only partially address the issue and have been of limited practical importance since they are rarely used. This lack of internal structuring is a large downside for use in a pervasive application since applications generally have to be hard coded to work with specific file formats. File systems also have poorly defined (or at least marginally useful) guarantees with respect to concurrent access. A final issue with file systems is that if a common naming hierarchy is needed across machines, they generally require continuous connectivity to a server (with an exception being Coda). One final advantage is that file systems tend to be fairly lightweight.

- *Databases.* Conventional relational databases have proven themselves to be a very efficient and effective way of storing and retrieving large amounts of table based data. One problem with using a conventional databases in a pervasive system is that there is generally no innate organization of tables beyond simple names. Database operations are focused on managing the data within a store and not in managing the logical stores themselves. Although remote access to databases can be dealt with using a client-server architecture, movement of databases between machines is generally not a built in operation. A second issue with using a database for a pervasive application is that although relational databases are very good at storing large amounts of fixed-schema data, and object databases are reasonable at storing data with slightly more relaxed schemas, databases generally are not good at storing and retrieving arbitrary heterogeneous data. Although there are many projects and standards attempting to map semi-structured data (generally XML) to and from relational databases[10] [3], such systems generally require that the data adhere to a fixed schema, which we consider unacceptable for pervasive computing. One strength of conventional databases for pervasive applications is that they provide very good semantics for managing concurrent access to data through the use of transactions and atomic record updates. Likewise, they provide a flexible and convenient means of accessing data through queries. Although conventional databases per se may not fulfill our needs, there are interesting systems such as Lore[8] which have explored taking a semi-structured approach to databases.
- *Distributed object systems.* In distributed object systems such as Globe[12], data and functionality are encapsulated by the same basic abstraction: the object. From a pervasive computing perspective, one very desirable property of this equivalence is that managing and moving stored data can be accomplished through the same mechanisms that are used to manage and move code. A second advantage is that such systems they allow any sort of data to be stored since any object can be treated as stored data. Unfortunately, the tight coupling of code and data also has strong drawbacks for pervasive computing. In particular, this coupling makes it hard for one program to even locate data created by another program since the interface and semantics of the data storage are application specific. Additionally, object oriented design encourages a style of programming in which data is not directly exposed: the functionality expected to be needed for dealing with the data is exposed. Although this is often a useful model for internal program data, it lacks flexibility in a pervasive computing world where a key expectation is that data will be accessed by many applications, often in new or unexpected ways. Finally, not having a fixed storage interface makes extending or enhancing storage to add properties such as replication difficult since there is no a single interface to be extended: any specialization needs to be repeated for each application.
- *Tuple Spaces.* These systems include the prototypical Linda system[4] as well as more modern variants such as IBM TSpaces[14]. Tuple spaces were originally developed as a communications and rendezvous

abstraction in which records with arbitrary ordered, and sometimes named or typed, fields can be inserted into and taken out of a uniform store. They generally support read operations parameterized by a pattern of the type of record to read. Although originally developed for communication, such systems can also be used for storage. One of the strengths of the tuple space model for pervasive storage is that it provides good support for storing and accessing heterogenous data. Any sort of data can be placed in the store. One downside to Tuple Spaces is that traditional Tuple Spaces such as Linda provide no structuring for the stores: the Linda Space is viewed as a single space pervading the world. Although in some ways attractive from a theoretical level, this is impossible to realize in reality. This disconnect between reality and model makes it hard to explicitly manage the grouping of data. TSpaces expands this model a bit by allowing named stores and by making the physical location of the stores visible to the application, but still provides no real method of organizing and maintaining the stores. These systems also tends to be central-server specific, with migration of data and disconnected operation being unsupported operations. A second downside to most current implementations is that they tend to be optimized for communication uses rather than for storage use. In particular, the underlying structure of most such systems is heavily dependent on in-core indexing structures, making them difficult to directly use as a model for a disk based system. One very nice feature of a Linda style system for pervasive computing is that it provides a very simple interface with only a few commands which could easily be extended or modified. Unfortunately many modern tuple systems such as TSpaces seem to have suffered from extreme feature creep (the TupleSpace class in TSpaces has 16 constructors and 105 other functions or variations of functions) and would be challenging to reimplement, trap, or extend.

Overall we felt that the Tuple Space design provided the best starting point from our design. We have, however, deviated from the basic model, in some places quite significantly, in order to address the slightly different needs of pervasive computing.

## 4 Design

From the outset, we wanted storage to be tightly integrated into our system as a first class object. Our architecture borrows heavily from conventional tuple store systems for the actual data storage and from file systems and nested processes for the organization of this storage. Our system is composed of a set of entities, which we call *Environments*, that hold data *tuples*. There are two relatively independent parts of our design: the management of Environments and the management of tuples within Environments. The organization and management of the Environments will be discussed first, followed by a discussion of the operations that can be carried out within an Environment.

### 4.1 Environment Hierarchy

The basic organizational abstraction in *one.world* is the *Environment*. An Environment contains three types of things: Components which are running code, Tuples which are permanently stored in the environment, and other Environments in a tree structure. Each *one.world* machine has a root environment which holds the kernel and is an ancestor of all environments on that machine. Each Environment exists on one and only one machine, with Environment hierarchies on different machines being disjoint. Code in an Environment by default has full control over its own environment and all child environments, but by default has no power over its ancestors. Any request by code within an Environment to perform an operation on its environment or on any of its descendant environments is filtered up the environment hierarchy, with any ancestor of the requesting Environment being allowed to deny or modify the request.

In addition to the storage API itself, there is an API for managing the stores. Though rather straight forward, its existence is critical and is one of the strong distinguishing marks between the *one.world* storage system and traditional tuple stores or databases. Environments are named through the use of textual path names relative to the current environment. Naming with respect to the current environment allows environments to be moved without the need to maintain the entire namespace. The storage management operations applicable to environments are *Copy*, *Move*, and *Bind*. *Copy* and *Move* copy or move an environment subtree (including both executing code and data) from one part of a nodes hierarchy to another part

of the hierarchy, or even from one machine to another machine. This operation gives us the tight coupling and management symmetry of code and data available in a distributed object system without the downside of making it difficult to differentiate the code and data when necessary. The final operation, *Bind*, creates a binding to an Environment, allowing us to read and write tuples from that Environment. In keeping with the *one.world* philosophy, store bindings are leased. This lease serves two purposes. First, it provides safety by allowing resources to be reclaimed if an application goes away either by exiting or by moving to another node. Second, it provides a well-defined mechanism through which change in the environment can be exposed to the application.

When an environment is moved or is deleted, all external bindings to that environment's tuple store are broken, and all bindings from within that environment are also broken. This is accomplished by canceling the lease on storage. The breakage is silent, with the client only discovering the breakage when it attempts a tuple store operation or an operation on the lease. Rebinding, if desired, must be performed by the applications. The decision to require application intervention in this case is purposeful: it exposes our belief that only the application itself can know what the correct action to take is when a resource goes away.

There is no built in support for accessing stores on remote nodes, however this can be easily added using some form of user supplied server. Most of the required work to support remote access involves complex domain specific policy issues that seem better addressed outside the kernel.

## 4.2 Store operations

The API was designed to support two uses: tuple storage (being discussed here) and tuple communication, though each use was expected to have its own implementation and slightly different semantics. Motivated by the success of other systems that use common abstraction for storage and communication (Unix file descriptors in particular) we felt this was an achievable goal. The store API is divided into a set of basic operations supported both by storage and communication, as well as an API extension supported only by storage. The basic operations are *Read*, *Write*, and *Listen*. The extended storage specific API adds *Delete* and *Query*. The read, write, delete and query operations can be grouped by transactions (currently unimplemented).

The meaning of the operations are as follows:

- *Write(Tuple)* Writes a tuple to the store. If the store already contains a tuple with the Id in this new Tuple, the old tuple is overwritten. For communications use, after all the listeners and readers on the receiver side have been checked to see whether they match this tuple, the tuple is discarded.
- *Read(Pattern)* A pattern specifying the tuple to retrieve is given. The store returns some tuple matching this pattern(if any exists). A timeout is given to facilitate communication uses where the read can wait for a write.
- *Listen(Pattern)* A pattern specifying tuples to return is given. The store echos any future writes of tuples matching this pattern to the requester. Listen requests are leased and must be periodically renewed.
- *Delete(Id)* A Guid representing the tuple to be removed is specified. If a tuple with this Id exists in the store it is removed and discarded.
- *Query(Pattern)* A pattern for tuples to match is specified. A leased event handler that can be used for iterating through all matching tuples is returned. The iterator has one operation: get next element. There is no guarantee that the returned tuples will represent any snapshot of the store, but it is guaranteed that if no changes to matching tuples are made to the store during the iteration all matching elements will be returned, and in any case that no element will be returned more than once.

Each of the input operations can also return only the Ids of the matching tuples instead of the entire tuples. Query returns only a single matching element at a time for two reasons: first, this allows a query implementation in which the memory requirement is that only as much memory as the largest tuple is needed, rather than requiring memory as large as all tuples. This will be important on small devices. Second, this

allows an optimized interface in which matches are returned as they are found, rather than having to wait for all matches to be made.

*Read*, *Listen*, and *Query* all rely on a common query system. This system is designed to allow for basic queries on the values and structures of tuples, but is NOT intended to provide the functionality of a full database query language. It is somewhat more powerful than pattern matching systems in Linda or TSpaces, but still allows a simple implementation.

The query system allows for the following types of queries:

- Field Equality - A field is exactly equal to a supplied value.
- Numeric ordering - An integer for floating point field is greater than or less than a supplied value.
- String contents - A string starts with, ends with, or contains a supplied substring.
- Tuple type - A tuple is of a specified type.
- Tuple subtype - A tuple is a subtype of a specified type.
- Field declared type - The declared type of a tuple field is of a specified type.
- Field declared subtype - The declared type of a tuple field is a subtype of a specified type.
- Field actual type - The actual object stored in a field is of the specified type (particularly useful for untyped tuples or tuples with fields typed Object so they can hold any value).
- Logical operations - Any of the above queries may be combined using *and*, *or*, and *not*.

A query in our system might do the following: Match tuples of type “ContactTuple” with an “age” field of type Integer, where “age” is greater than 21, and where the “FirstName” field is not equal to “Eric”. Although far less powerful than database query systems such as SQL, these queries are significantly more powerful than Linda queries, which only allow simple pattern matching. One advantage of our simplicity is that our queries are relatively trivial to parse, and their effects are fairly easy to understand, making it easy for a third party to extend and override these calls. Note that our language only allows queries that depend on the value or structure of a single tuple (no joins, sorting, or “max” queries), making it slightly easier to implement and, more importantly, making it a useful system for operations such as *Listen* which innately operate on a single tuple rather than on a global store of tuples. We expect that apart from its use in storage, our query language will see wide use in user-level tuple filters.

### 4.3 Missing operations

For someone familiar with standard tuple stores, there is one glaringly obvious omission in our system: *Take*. In a conventional tuple space, *Take* is given a tuple pattern and returns any tuple matching that pattern while simultaneously removing the tuple from the store. Also missing is a *Delete* operation parameterized on a pattern. The reason for these omissions was not difficulty of implementation or even a feeling that they were useless, but was in fact that these operations make extending and overriding the storage API more difficult since they are state changing operations where the state being changed is not obvious on the inbound path. More specifically, a *Take* operation changes the store by removing a tuple, but which tuple will be removed cannot be predicted prior to sending the request to the store. This makes layering operations such as logging on top of the store more difficult since both the inbound and outbound paths must be trapped. Additionally, if such a layered application wishes to ensure that two concurrent operations do not affect the same tuple (for instance, when replicating data, it is important to know the order of operations on a given tuple), operations such as *Take* require the entire store to be locked rather than only a single Id.

Although *Take* is a very important operation in normal tuple store systems, the use of a transaction with a *Read* and a *Delete* can replace it, albeit with a slightly higher cost due to conflicts. In any case, it seems unlikely to be a key an operation in our system. *Take* is traditionally important since Linda systems primarily used stores for communication. For that use, the ability to efficiently read and remove a tuple is

crucial since it is used as the mechanism for event handoff. In our store, which is designed primarily for long-term storage, it seems less likely that read/delete will be common.

One final argument against using *Take* in any system such as ours that is based on asynchronous events with “best effort, at most once” delivery semantics is that it is inherently unsafe: if the *Take* request makes it to the store, but the response is dropped either due to a queue overflow or due to the requesting environment having moved to another node, the tuple being “taken” is lost forever.

## 5 Implementation

At present, the storage API has been implemented with the exception of transactions. The storage system is implemented on top of Sleepycat’s Berkeley DB (BDB)[11] system. Although the use of BDB is not integral to our design, it provides many useful primitives that we would otherwise have needed to implement ourselves. BDB is in fact not a database per se: it is a library that provides primitives useful for implementing various database-like systems. BDB offers sorted tables of keyed records, with arbitrary length binary keys mapped to opaque, arbitrary length binary entries. Transactions and reliability in the case of failure are provided through the use of logging. BDB is a C library which runs on Linux and Windows and which offers a Java API.

The bulk of the current implementation is related to keeping track of the on-disk environment hierarchy as well as maintaining the set of all open bindings to storage. The actual tuple operation code is relatively small. The basic mapping is that a single BDB table is created per environment, with each tuple being saved in standard Java serialized form keyed by its Id field. In general, read requests currently require a sequential read and comparison of each tuple in the entire store until a matching tuple is found, but queries specifying a specific Guid do a keyed lookup instead. In the near future our system will be extended to allow more sophisticated indexing and retrieval.

In addition to the per-environment stores, two BDB tables are used to maintain the environment hierarchy. One table maintains a mapping of an environment to its children (a single entry per child) and the second table maps an environment to its protection domain, its parent Environment, and its textual name. The table for each database is stored in a directory tree derived from the Guid of the environment, limiting the total number of BDB backing files in a given directory since many filesystems perform badly when operating on large directories. Currently a BDB backing table is created initially with every environment, but an obvious future performance optimization will be to make this creation lazy since many environments may only hold code and not tuples. All stores share a common BDB environment and log, allowing transactions to span tables.

The tuple store is concurrency safe. Multiple storage operations can be conducted simultaneously, with Berkeley DB locking mediating access to the stores. For management of Environments, a combination of BDB locking and Java synchronization are used to ensure safety. Locking is performed at the finest practical granularity, generally at the single lock per store level to prevent lock contention between applications accessing different stores.

We ensure ACID properties through the use of per-operation transactions implemented using standard BDB transactions. Operations on environments themselves can’t be directly transactionally protected, so a prepare/commit system is implemented indirectly using transactionally protected meta-data writes in tables that allow the tuple store code to clean up properly after a crash during environment management. Tuple store moves between systems are dealt with using a prepare/commit/abort which ensures that an environment will not be deleted until the remote system has acknowledged the move. Although this ensures that an environment will not be lost, it is possible that a particularly unluckily timed network failure will cause a move to result in two copies of an environment tree.

## 6 Benchmarks

Basic performance measurements were conducted to determine the overhead above raw Berkeley DB (BDB) and the absolute performance compared to IBM TSpaces. The benchmarks measured the read, write, and query latency of Berkeley DB, *one.world*, and TSpaces. *one.world* used an event based client while the

Test	BDB	Serial BDB	<i>one.world</i>
Read	0.24 ms	0.88 ms	1.23ms
Write	25 ms	25 ms	25ms
Query (all)	44 ms	640 ms	860ms

Figure 1: Per-operation latency for primitives store operations. BDB performance includes only the actual IO performance when operating on pre-serialized tuples. Serial BDB performance also includes the cost of serialization for writes and the cost of deserialization for reads or queries.

Berkeley DB and TSpaces tests used a conventional threaded client. These benchmarks are neither claimed nor expected to highlight real world performance, but instead are meant to demonstrate the reasonableness of the implementation.

A Pentium III 800 with 256mb of ram running Windows 2000 using the Sun jdk1.3.0 JVM was used. Networking tests used a 100Mbit ethernet. Berkeley DB 3.2.9 with btree indexing was used for the BDB tests. TSpaces 2.1.2 was used for the TSpaces tests. The data used for most store tests is a PIM contact tuple designed to allow a direct comparison with TSpaces. In particular, it is a flat tuple with all data fields being directly in the tuple rather than in sub tuples since TSpaces is designed around such a model. Each tuple had an average of 18.5 entries including generic tuple fields, with the average size of a serialized tuple being 850 bytes. A store size of 1000 tuples was used for all tests. This test was chosen because it seemed a reasonable size for a large PIM contact database and because it is hopefully large enough that inefficient algorithms will be readily apparent. Both the size and number of tuples seem reasonable for a contact database. Using relatively small tuples and a moderate databases also has the property that overhead should show up strongly since actual disk I/O will likely not be a bottleneck due to caching.

For the Read tests, the random tuples were first written into the store. The Ids were stored in an internal data structure and the time required to sequentially read 1000 randomly chosen tuples using Id keyed reads was measured. For the BDB tests, numbers are reported both with and without a deserialization of each read tuple. For the Write tests, 1000 random tuples were first created and stored in memory to avoid measuring object creation time. The time taken to sequentially write these tuples into an empty store in random order was then measured. For the BDB tests, numbers are reported both for writing pre-serialized tuples and for writing tuples serialized immediately before each write. For the Query All tests, the random tuples were first written into the store. The time taken to issue a query matching all tuples and to iterate through all of these tuples was measured. In the BDB case, this was accomplished using a BDB cursor to simply iterate through all entries in the database. Again, BDB results are presented with and without deserialization of read tuples.

For each of these benchmarks, the test was repeated 100 times, with the average latency per operation of these trials being reported. All of the tests were conducted with a warm cache (the first run was discarded), so given the size of the data it is expected that most data was cached except in the case of writes, where *one.world* semantics forced disk synchronization.

Figure 1 measures the performance of the basic *one.world* storage operations as well as the underlying BDB performance on these tests to measure our overhead. These tests show reasonable overhead, with a 40% read overhead compared to raw BDB plus serialization. The test does, however, indicate that serialization is a major expense and that investigating a less expensive method of serialization might be worthwhile. There is some hope of making headway in this endeavor for our common case (a tuple with only public fields containing strings and integers) since this is significantly simpler than the Java common case of a random Java object. The write latency was constant in each case since the choice of durable writes forced a synchronization to disk on each write. With durability in the case of failures removed, but the other ACID properties maintained, write latency drops to under 1ms.

We compared our storage latency to IBM TSpaces, an in-memory tuple store with checkpoints for durability, to ensure that our absolute performance is reasonable. For this test we used the Read and Write benchmarks mentioned in the first section as well as a Query test which utilized pattern matching. For the Query test, 1000 random tuples were first written into the store. The time taken to issue and iterate through a query returning all matches for a randomly selected first name was used. On average, there were 23 matches per query.

Test	<i>one.world</i>	TSpaces
Read	1.2 ms	2.9 ms
Write	25 ms	4.5 ms
Query (23 matches)	690 ms	18 ms

Figure 2: The performance of *one.world* versus the performance of IBM TSpaces.

Tuple count	Tuple size	Move Latency
0	NA	0.39 s
100	100 byte	0.70s
1000	100 byte	2.7 s
10000	100 byte	23 s

Figure 3: The latency to move a *one.world* environment.

All TSpaces tests, including writes, were to RAM since TSpaces is a memory based store and TSpaces periodic disk checkpointing was not used. The TSpaces server was run on the same machine and in the same JVM as the benchmarking software. Initially, communication with the server was conducted using the TSpaces local JVM mode, but there appeared to be a bug in this mode of operation that caused operations to slow dramatically over time. Using the normal communication methods this was not an issue, and the normal communication mode was thus used. Timings of the initial operations in the local communication method appeared very similar to those in normal server mode so we do not believe this had a substantial impact on performance.

The results of these tests are shown in Figure 2. Due to differences between the systems it’s not possible to directly rank ourselves against TSpaces, but we are in the same performance bracket for reads and writes, with writes being slowed by our choice of durable writes. An experiment with durability turned off indicated that writes would require less than 1ms if we were to ease this constraint. The one major performance issue with the *one.world* results is field match queries. The slowness of this operation was expected since we currently index only on Ids, requiring a traversal of the entire store. TSpaces uses a fully indexed memory structure, so it can avoid this traversal.

Our final benchmark measured the latency of our environment move primitive using a small application that moves itself and its environment in a loop around a set of machines. Small tuples carrying 100 byte arrays were placed in the environments to emphasize any overhead in our system by deemphasizing raw data transfer performance. The test was conducted with stores containing varying numbers of tuples. The average cost of a move was benchmarked by measuring the average cost of a move during a 100 loops around three hosts. The reported value is the average cost per move. The results are tabulated in Figure 3. This benchmark is hard to evaluate since there is no clear point of comparison, however the performance seems reasonable. A raw move occurs at speeds seemingly acceptable by human standards (an important metric for migration). Additionally, if the move latency for the zero tuple case is subtracted from the latencies for the other three cases, the remaining latency per tuple remains relatively constant for all three tests (3.1ms for the 100 tuple case, 2.3ms for the 1000 tuple case, and 2.3ms for the 10000 tuple case). This indicates that movement latency is roughly linear in the number of tuples within the environment as we would hope.

## 7 Experience

Beyond the performance evaluation, we have begun using the system and can offer some preliminary observations. Currently there is one major non-core user of tuple storage in our set of core packages (a replication layer) and several small application users. Although we are still early in the use of our architecture, the creation of some small initial systems using it (as well as the implementation of the system itself) have highlighted both good and properties of our system. Although the basic design for the *one.world* tuple storage system seems sound, experience implementing and using the storage system has highlighted several areas that could use work. This experience will be related along three axis: 1) the overall utility of our storage

features, 2) minor flaws or annoyances noted in basic API features, and 3) more fundamental issues related to our event delivery semantics.

## 7.1 Utility of abstractions

Our work creating a replication layer for *one.world* indicated that the basic hierarchical design is sound and that our interface is in fact readily extensible. In order to log writes, a parent replication environment simply captures the storage binding request of the application and returns a wrapped handler which logs store modifying operations before sending them on to the real store. *Read*, *Query*, and *Listen* requests are passed straight through the wrapper and can be executed directly by the underlying store. With very little code, it was possible to create a service that intercepted the operations of a pre-existing service to provide new functionality.

Likewise, the ability of our system to cope with heterogeneous data sources proved useful. Specifically, an undergraduate class project, *one.box*, using this system was designed around the idea of providing general purpose storage, filtering, and routing of heterogeneous data types and found the capability to store and access arbitrary data types very useful. When implementing a prototype of PIM system, the ability to deal with anonymous heterogeneous types was very useful. Although the stored data largely followed a fixed schema, the ability to query and inspect unknown data types allowed the use of a generic tuple interface to read and modify arbitrary contact tuples.

The ability to group storage and applications together in an environment and to migrate these grouped entities simply by moving the environment has proven highly profitable. In particular, we found that the pattern of a mover application and environment which encapsulate migration logic to be useful. By embedding stores containing our application data as well as the applications themselves within this mover environment, we can migrate arbitrary applications and their data using logic unforeseen when the generic application was written (or even initially executed).

In all cases, the largest amount of work was dealing with failure cases. In particular, implementing timeouts and retries to recover from lost events is tedious and time consuming. This issue is not specific to storage and is present throughout our applications. The *one.world* group is designing programmatic patterns to allow application designers to reuse error handling code to ease this burden.

## 7.2 API features

Although we feel that our system generally showed itself to be useful, there are some minor issues related to the API that we discovered. The majority of these issues can be classified as minor issues and will be easily remedied in releases in the near future.

As expected, the two major unimplemented features in our system (transactions and indexed queries) would have been useful. Our experience showed that the lack of indexing was particularly a problem. The poor performance of query based *Read* calls was a significant issue for replication, requiring a redesign in which almost all accesses to the logged store are accomplished through Id keyed reads rather than queries. The lack of transactions was also an issue, but could largely be worked around by making the simplifying assumption that only a single application was accessing the store at a time and then using internal synchronization. This assumption is largely reasonable for replication since its internal stores are very application specific, but this assumption will become less reasonable as more complicated applications, such as calendaring systems, are developed that will share state used by many application in a single store.

One operation that we found to be incomplete in its current incarnation is *Listen*. In particular, it reports writes to the store but does not report deletes. This is sufficient for communication (there is no delete operation) as well as some limited storage applications, but is not sufficient for storage users who want to use *Listen* to maintain an outside representation of the store state. For instance, a small PIM contact database was written to test the replication layer. In this application, a window is presented to the user showing all contacts currently in the database. While this window is open, replication may synchronize the contact database with a master server, updating the contents of the store and the application would like some way to keep its window up to date. *Listen* is the natural operation to accomplish this, however it is only able to deal with the addition or modification of an entry: it cannot detect the deletion of entries. The basic issue is that we initially viewed the *Listen* in the conventional Tuple Space manner as a method for

detecting newly arrived messages. This model is not quite the model that turned out to be most useful for applications. Applications such as our PIM manager do not view *Listen* as a way of detecting new data, but instead view it as a mechanism for maintaining an image of the state of the store. This subtle change of viewpoint leads us to instead view *Listen* as a mechanism for reporting any state changing operations on a store. As a consequence of this experience, the API will be modified to allow reporting of *Deletes*.

A second, much more minor, API issue involved lease management. In particular, maintaining leases for *Listen* and *Query* is tedious. Although leases are useful for both operations as a common mechanism for releasing resources, in these case lease renewal is not necessary for safety since these leases are canceled when the overall storage lease expires. It in general would seem to be a good idea to allow non-expiring leases in cases where a lease is dependent on an already existing lease. Although this is in some ways a minor detail, decreasing programmer burden wherever it does not interfere with other objectives seems critical.

Although the query language generally worked well, there were some notable deficiencies. We wish to avoid the temptation of recreating SQL, but there are a few additions that seem vital. In particular, while designing a data model for a PIM application we realized that the current API provides no means of matching based on whether any element of an array field contains a specific value. This turns out to be a vital query for PIM usage. For example, a good PIM application will allow any number of email addresses to be recorded for any individual. One natural way to support this is simply to place an array of strings in the Email field. Unfortunately, with the current query language this means that email addresses cannot be searched. The solution being implemented is simply the straightforward solution of allowing arrays to be searched through the use of an array element wildcard.

Finally, we had one performance issue that is not due directly to our API but instead to our ACID semantics for written tuples. In particular, we guarantee that once an application has received a response for a write, this write will be maintained even if the system crashes. Unfortunately this durability comes at a significant performance cost: each write must be synchronized to the disk before replying, causing a 25ms latency per write on our test system. Although performance is not the the driving force behind our design, architectural decisions that force significant performance penalties (25x in this case) must be carefully considered. Implementing userspace transactions which allow multiple writes to be grouped into a single synchronized write will help, but this is not a complete solution. Likewise, there are some implementation tricks that can help in some cases, but the best solution in many cases might be an optional easing of the durability constraint. This option will be investigated.

### 7.3 Event delivery

The final set of experiences revolve around a more fundamental aspect of our system: the “best effort, at most once” event delivery semantics which guarantee that an event will never be received twice but may not be received and may in some extreme cases may be dropped without even informing the sender. This property of *one.world* make the use of a straightforward API significantly less straightforward in practice. These semantics are hard to avoid in this sort of system. For remote communication, this semantic model is a natural fit. For local communication, event queues can overflow or safety and desired semantics may make it impossible to retry sends forever. Although we expect dropped messages to not be the common case, it is important to provide an API that deals gracefully with even the exceptional cases.

One operation in which this becomes an issue is the *Query* command. In particular, the *Query* interface provides no way to guarantee that all tuples will be seen in the presence of dropped messages. Worse than this, it is in fact not possible to be sure whether or not we in fact saw all of the tuples if any drops do occur, since we cannot tell whether it was a request or a response that was dropped. It should be emphasized that if each request does result in a response we are guaranteed that we have seen all messages. This first property(that not all tuples will be seen) is in fact an issue for any sort of request/response interaction in which the operation is not idempotent. The second is an issue for any operation which is not idempotent and which gives no information differentiating sequential invocations. The importance of idempotent operations is well known in systems with loss and is the reason that most NFS operations were designed to be idempotent [9].

In the *one.world* storage system, the *Read* and *Listen* requests are idempotent. *Write* requests are largely idempotent except for the fact that repeated *Write* request will create repeated listen responses. *Delete* is

largely idempotent except that a second delete for a given tuple will return an error stating that the tuple was not found. The *Query* request itself is idempotent, but unfortunately the iterator *next element* operation is not idempotent: repeated iterator requests will by design return different elements.

The detection of loss in this instance could be fixed through the use of sequence number tagging. By sequentially tagging returned value, a lost request can be differentiated from a lost response by watching for skipped sequence numbers. Providing reliability seems to require a more substantial modification of the interface. In particular, the most straightforward solution is simply to make the operation idempotent. One obvious way to do this is to allow the iterator to randomly access elements. To achieve repeatable results, this requires that we either buffer already read elements or that these elements be locked until the iterator is closed to allow repeatable scanning. A similar method that avoids these downsides is to allow the reader to specify the next sequence number to read, but to restrict this number to be either the last element returned or the next element. This allows requests to be repeated while not requiring much additional storage or work on the part of the server. Although this allows reliable operation, it has the downside of not allowing parallel iterator requests to be issued since we cannot be guaranteed of the request delivery order. Without buffering a larger number of elements there seems to be no straightforward way to accomplish the dual goals of reliability and parallelism.

There is a similar issue with our Listen operation, with similar but subtly different properties. As with *Query*, the *Listen* interface provides no guarantee all messages will be delivered. Unlike the case of *Query*, it is unclear that perfect reliability is obtainable or desirable. In particular, perfect reliability requires either that all *Writes* to a store be throttled to the pace of the slowest *Listen* receiver or that unbounded storage be available. A direct implementation of the former is likely unacceptable for reliability reasons (malicious or broken programs could use it to institute denial of service attacks), and the latter is likewise untenable. This unreliability unfortunately makes using *Listens* to maintain a list of changes to storage somewhat problematic. The two strategies for dealing with this seem to be either to accept this deficiency and use Listen only to maintain soft state about the store, or to provide some mechanism to allow applications to detect that they have missed *Listen* events, even if we cannot tell them what these events were.

As before, the use of sequence numbers to sequentially enumerate responses seems a viable alternative for detecting loss, however the actual interpretation of these numbers is non-trivial due to the possible out of order arrival of listen events at the receiver. Sequence numbering listen responses also requires synchronized access and modification of a sequence number counter on each write which would otherwise be unnecessary, though this seems unlikely to be an undue burden.

In addition to the issue of detecting losses, there is a second issue for some users of *Listen*: currently the ordering of writes to the same tuple cannot be determined. This makes it impossible to use *Listen* to maintain a completely consistent view of the tuple store even without losses. Assuming that sequence number allocation is done carefully (sequence numbers for writes involving the same tuples are issued in the same order as their transactions commit), this also nicely solves the problem of disambiguating multiple modifications to a single tuple.

## 8 Future work

Although the basic model and API seem sound, there are still areas of the implementation and the API that need to be improved. Several modifications to the API are currently being planned. In particular, notification of deletes to listeners will be incorporated. Additionally, the query language will be modified to provide a bit more flexibility in matching elements of arrays. Modifying the Query and Listen interfaces to return sequence numbers to allow applications to detect failures seems relatively straightforward and worthwhile and may be implemented at some stage.

One of the more important areas currently being addressed is indexing. This is a somewhat harder problem than that encountered in standard relational databases due to the lack of a fixed tuple schema. We plan to use auxiliary index tables and choose the index fields based on hinting. At a low-level, indexing solutions similar to those implemented in Lore[8] will probably be applied, with the selection of which indices to create being controlled by an explicit hinting system such as that in PStore [6].

Transactional support is defined but unimplemented in the current architecture. The expectation is for the transactional system to be a straightforward local transaction system leveraging the Berkeley DB

supplied transactions. This is, unfortunately, not completely straightforward at an implementation level due to a locking design that is quite common in threaded architectures, but not well suited to event systems. In particular, a Berkeley DB transaction may only be used within a single thread, and a single thread cannot safely use multiple transactions. These requirements do not mesh well with an event queue design where the queue is served by a set of threads. At a somewhat higher level, there are also some issues with how best to deal with deadlock detection in a system where blocking is not the common mode of operation.

## 9 Conclusion

*one.world* attempts to design a highly integrated storage architecture well-suited to the needs of pervasive computing by combining desirable properties available in several existing storage architectures. Although certain rough edges have been discovered during initial usage, the basic model seems sound, with several features such as the mobility of stores being very useful. One strong lesson from this work is that closely examining the behavior under exceptional conditions during the initial design phase is very important. Additional application implementation work needs to be conducted to further evaluate the design decisions.

## 10 Acknowledgements

Most of the system and API definition was a collaborative effort between the *one.world* group members, with the initial environment hierarchy model largely defined by Robert Grimm. Much of the tuple filter implementation as well as some very initial work on the tuple store itself was done by Ben Hendrickson.

## References

- [1] BRAY, T., AND SPERBERG-MCQUEEN, C. Extensible markup language (xml). W3c working draft, W3C, November 1996.
- [2] DOUGLIS, F., AND OUSTERHOUT, J. Transparent process migration: Design alternatives and the sprite implementation. *Software Practice and Experience* 21, 8 (August 1991).
- [3] FLORESCU, D., AND KOSSMANN, D. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Tech. Rep. 3680, INRIA Rocquencourt, France, May 1999.
- [4] GELERNTER, D. Generative communication in linda. *ACM Transactions on Programming Languages and Systems* 7, 1 (1985), 80–112.
- [5] GRIMM, R., DAVIS, J., HENDRICKSON, B., ERIC LEMAR, A. M., SWANSON, S., ANDERSON, T., BERSHAD, B., BORRIELLO, G., GRIBBLE, S., AND WETHERALL, D. Systems directions for pervasive computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau, Germany* (May 2001).
- [6] GRIMM, R., SWIFT, M., AND LEVY, H. Revisiting structured storage: A transactional record store. Tech. Rep. UW-CSE-00-04-01, University of Washington, Department of Computer Science and Engineering, April 2000.
- [7] LEACH, P. J., AND SALZ, R. Uuids and guides. Internet-draft (expired), IETF, February 1998.
- [8] MCHUGH, J., ABITEBOUL, S., GOLDMAN, R., QUASS, D., AND WIDOM, J. Lore: A database management system for semistructured data. *SIGMOD Record* 26, 3 (September 1997), 54–66.
- [9] MICROSYSTEMS, S. Network filesystem specification. Tech. Rep. RFC 1094, IETF, March 1989.
- [10] SHANMUGASUNDARAM, J., TUFTE, K., HE, G., ZHANG, C., DEWITTT, D., AND NAUGHTON, J. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999* (September 1999), pp. 302–314.

- [11] SLEEPYCAT SOFTWARE, INC. Berkeley DB 3.2.9. <http://www.sleepycat.com>.
- [12] VAN STEEN, M., HOMBURG, P., AND TANENBAUM, A. Globe: A wide-area distributed system. *IEEE Concurrency* (January-March 1999), 70–78.
- [13] WEISER, M. The computer for the twenty-first century. *Scientific American* 265, 3 (September 1991), 94–104.
- [14] WYCKOFF, P., MCLAUGHRY, S., LEHMAN, T. J., AND FORD, D. A. T spaces. *IBM Systems Journal* 37, 3 (1998), 454–474.