

The Design of a Storage System for Pervasive Computing

A thick, horizontal yellow brushstroke with a textured, painterly appearance, extending across the width of the slide below the title.

Eric Lemar

Quals Presentation

May 25th 2001

Pervasive Computing



- ⌘ Computing power pervades our everyday environment.
- ⌘ Computing environment and computations are fluid
 - ☑ Network connectivity is common but limited
 - ☑ Computers appear and disappear
 - ☑ Multiple devices and programs will be used when completing a task

Purpose



- ⌘ Many interesting pervasive applications are heavily reliant on stored data
 - ☑ contact databases, scheduling, news, bus schedules, pervasive sensor capture

This work aims to develop a storage system addressing the specific needs of pervasive computing.

Pervasive Storage Requirements



⌘ Grouping of data

Aids security and mobility

⌘ Flexibility of storage

Need to use heterogeneous and unknown data

⌘ Concurrency safety

Applications will cooperate on tasks

⌘ Useful pervasive operations

For instance, query or migrate operations

Properties of interfaces



- ⌘ Universally available
- ⌘ Extendable
- ⌘ Compatible with asynchronous events and “best effort, at most once” semantics
- ⌘ Simple
- ⌘ Efficient

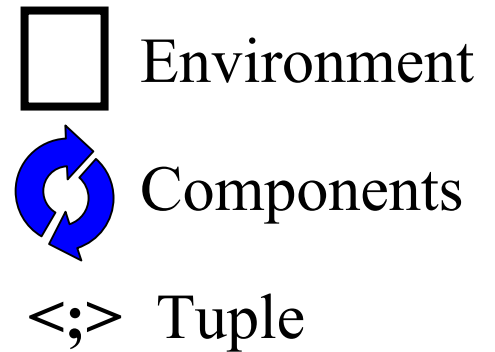
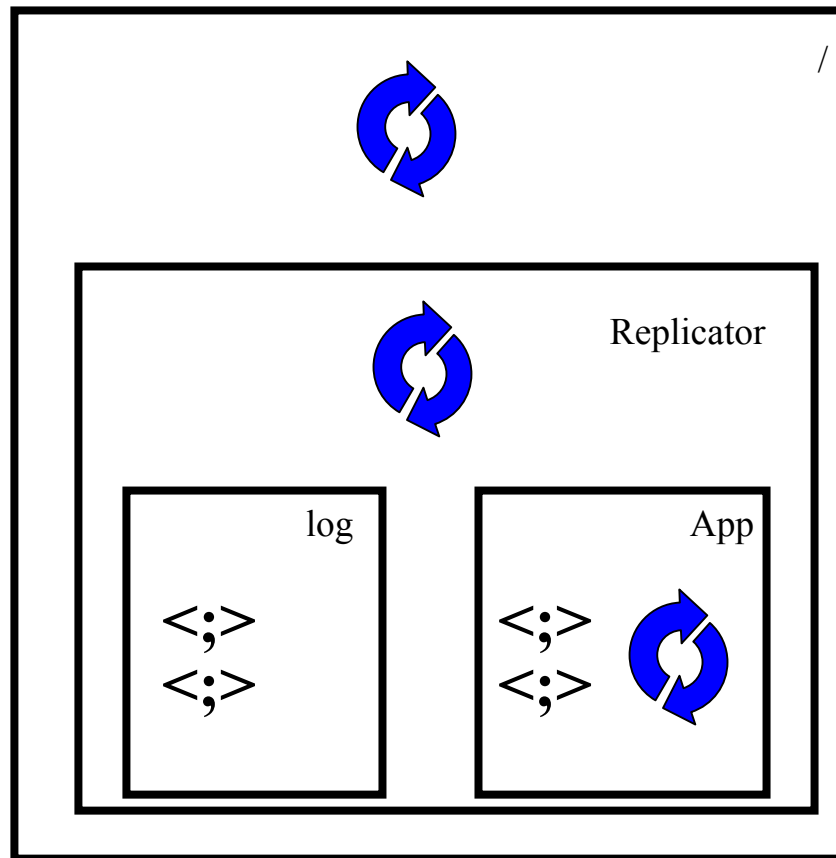
Data



Tuple

- ☒ Record with named and optionally typed fields
- ☒ Can contain nested sub-tuples
- ☒ Contains an Id field – A globally unique identifier

Environment Hierarchy



API



Two parts of the API

1. Environment Management
2. Tuple storage

Environment Management



⌘ Three operations

- ☑ Move

- ☑ Copy

- ☑ Bind

⌘ Filtered up the tree: parent environments can deny or modify the requests

- ☑ Component is allowed to access any environment below it in the environment tree

Bound stores



⌘ Store bindings take the form of leased handlers

- ☑ Receive an EventHandler to send storage requests to

- ☑ Lease contains a duration (non-infinite) and an EventHandler used to manage the lease

- ☑ Leases used to deal with change

 - ☑ Provide resource reclamation

 - ☑ Allow bindings to be broken

Store Operations



- ⌘ Write(Tuple)
- ⌘ Read(Filter)
- ⌘ Query(Filter)
- ⌘ Listen(Filter)
- ⌘ Delete(Id)

- ⌘ Optionally combine operations using transactions (unimplemented)

Note that there is no Take(Filter) operation

Filter Language



- ⌘ Tuple/Field Equality
- ⌘ Numeric Ordering
- ⌘ String value - starts with, ends with, contains
- ⌘ Tuple type or subtype
- ⌘ Field declared type, declared subtype, or actual type
- ⌘ Logical Operations (and, or, not)

Match tuples of type "ContactTuple" with an "age" field of type Integer, where "age" is greater than 21 and "FirstName" is not equal to "Eric"

Filter Language



- ⌘ All operations operate on a single tuple
 - ☑ No join, no sort, no “Max” queries
 - ☑ Makes it a useful language for filters

Implementation



- ⌘ Implemented on top of Sleepycat's Berkeley DB (BDB)
 - ☑ Toolkit for building database applications
 - ☑ Provides keyed/sorted storage
- ⌘ One backing database per environment
- ⌘ Most queries currently require sequential read and comparison of all tuples
- ⌘ Id keyed reads perform a direct lookup
- ⌘ Tuple operations are transactionally protected

Benchmarks



⌘ Latency measurements

- ☑ PIM contact tuple (average 860 bytes serialized)
- ☑ 1000 tuples per store
- ☑ Measure the average latency of sequential read, write, and query operations
- ☑ Meant to determine *one.world* overhead and reasonableness of implementation, not to model real usage patterns

Benchmarks

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

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

- ~70% The performance of Serialized BDB
- Java serialization is expensive
- Performance comparable to Tspaces
- Lack of indexing is problematic

What worked



⌘ Overrideable interface

- ☑ Allowed replication to be added to existing applications

⌘ Environment hierarchy

- ☑ Allowed applications/data to be migrated by a third party mover application.

⌘ Heterogeneous queryable stores

- ☑ Allowed generic tuple filters and editors

Interface issues



- ⌘ Lack of indexing a major bottleneck
- ⌘ Listen needs to report deletes
- ⌘ Durability semantics impose a significant performance penalty
- ⌘ Unnecessary lease management
- ⌘ Too much repeated code for binding and error handling

Query



⌘ Query

- ☑ Issue a pattern matching request
- ☑ Receive an iterator
- ☑ Send “Next Element” events to the iterator

⌘ But what if an event is lost?

- ☑ If the request is lost, merely redo it
- ☑ If the response is lost, we miss an element
- ☑ How do we know which happened?

Listen



⌘ Listen events can be lost

- ⊞ Makes it hard to maintain hard state
- ⊞ Can't even tell if we've missed an Event

⌘ How do we deal with this?

⊞ Detect losses

- ⊞ Add sequence numbers
- ⊞ Also lets us differentiate concurrent conflicting operations

⊞ Prevent losses

- ⊞ Need to ack listen events
- ⊞ We run at the speed of the slowest receiver

What is the problem



⌘ Query is not idempotent

- ☑ Repeated operations do not yield the same answer
- ☑ Query gives no distinguishing information

⌘ How can we fix this?

- ☑ Add sequence numbers so we can distinguish lost responses/requests
- ☑ Make query idempotent

Future work



⌘ Implement indexing

- ☑ Use secondary index tables
- ☑ Use hinting for index creation
- ☑ Look to Lore for inspiration

⌘ Implement local transactions

- ☑ Use Berkeley DB transactions

Recap



Grouping of data: Environment hierarchy

Flexibility: Semi-structured data

Concurrency safety: Transactions

Useful pervasive operations: Query, Migration

Universal availability: First class primitive

Extensibility: Simple, trappable interface

Conclusion



The basic abstractions appeared sound, however several details of the API need fine tuning both to add useful features and to correct deeper semantic problems.

More applications need to be written to verify the utility of our design.