ZOPE CORPORATION

Persistent Programming with ZODB

10th International Python Conference
Alexandria, Virginia
February 4, 2002

Jeremy Hylton and Barry Warsaw
{jeremy,barry}@zope.com

What is Persistence?

- Automatic management of object state; maintained across program invocation
- Frees programmer from writing explicit code to dump objects into files
- Allows programmer to focus on object model for application

ZODB Approach to Persistence

- Minimal impact on existing Python code (transparency)
- Serialization (pickle) to store objects
- Transactions to control updates
- Pluggable backend storages to write to disk

Alternatives to ZODB

- Many:
  - flat files, relational database, structured data (XML), BerkeleyDB, shelve
- Each has limitations
  - Seldom matches app object model
  - Limited expressiveness / supports few native types
  - Requires explicit app logic to read and write data

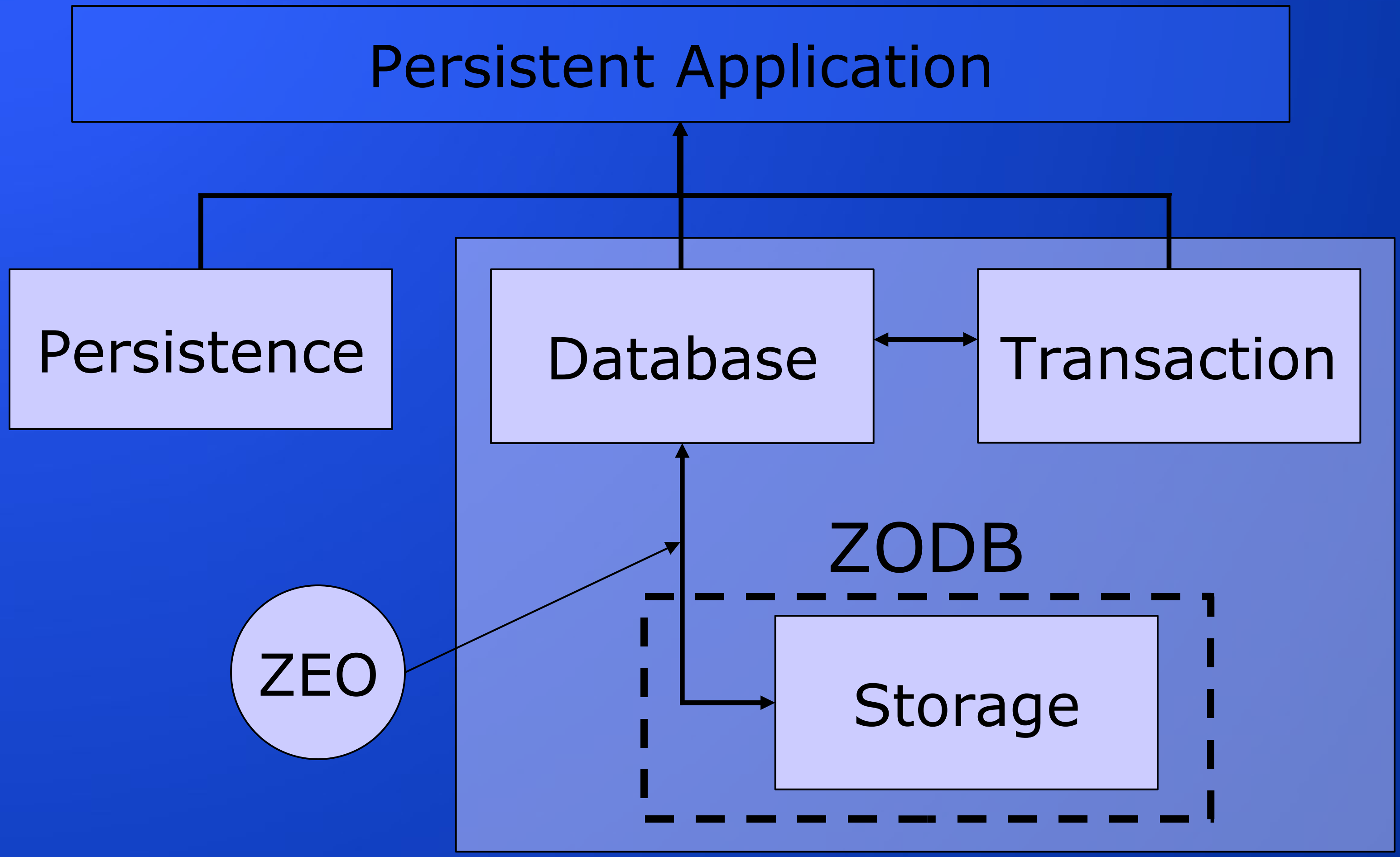
ZODB -- the Software

- Object database for Zope
  - Designed by Jim Fulton
  - Started as BoboPOS
- Extracted for non-Zope use
  - Andrew Kuchling
- Source release w/distutils from Zope Corp.
  - January 2002
- Wiki: http://www.zope.org/Wikis/ZODB
  - info central for ZODB

Software architecture

- StandaloneZODB packages
  - Persistence, ZODB, ZEO
  - ExtensionClass, sundry utilities
- ZODB contains
  - DB, Connection
  - Several storages
- Compatibility
  - Runs with Python 2.0 and higher
  - ExtensionClass has some limitations
    - No cycle GC, no weak refs, …

ZODB Architecture (1)



Public Components

- Components with public APIs
  - Database
    - allows application to open connections
    - connection: app interface for accessing objects
  - Transaction:
    - app interface for making changes permanent
  - Persistent base class
    - Logically distinction from ZODB

Internal Components

- Storage
  - manage persistent representation on disk
- ZEO
  - Share storage among multiple processes, machines

Future ZODB Architecture

- ZODB4 will isolate components
  - Persistent, Transaction interfaces separate
  - Database, Storage stay in ZODB
- Advantages
  - Allows other databases, e.g. object-relational mapping
  - Use Persistence, Transaction without ZODB

Key ZODB Concepts

- Persistence by reachability
- Transactions
- Resource management
  - Multiple threads
  - Memory and caching

Persistence by Reachability

- All objects reachable from root stored in database
  - Root mapping provided by database
- Each persistent object stored independently
  - use pickle
  - all non-persistent attributes included
  - customize with __getstate__()

Transactions

- Coordinate update of objects
  - Modified objects associated with transaction
  - Commit makes modification persistent
  - Abort reverts to previous state
- Means to cope with failure
  - Conflicting updates
  - Something goes wrong with system

Resource Management

- Threads
  - One thread per transaction
  - Controlled sharing via transactions
- Memory
  - Database contains many objects
    - Too many to fit in memory
  - Objects moved in and out of memory
    - ZODB manages this automatically
    - Knobs exposed to applications

Writing Persistent Applications

- This section will:
  - Introduce a simple application
  - Show how to make it persistent

(Very) Simple Group Calendar

- Calendar which can display a whole month, or a single day, with events
- Can create new appointments with rendezvous information
- Can invite people to an appointment

Group Calendar Objects

- `Calendar` – holds appointments keyed by subject and date (sorted)
- `Person` – has a name; later updated to username, realname
- `Appointment` – holds date, duration, subject, location, list of participants
- (a driver script)

Required imports

- Applications **must** import ZODB first, either explicitly or implicitly through a package reference
- Importing ZODB has side-effects (this will be fixed in ZODB4).

```
import ZODB
from ZODB.DB import DB
from ZODB.FileStorage import FileStorage
from BTrees.OOBTree import OOBTree
# Works as side-effect of importing ZODB above
from Persistence import Persistent
```

Creating persistent classes

- All persistent classes must inherit from Persistence.Persistent

```
from Persistence import Persistent
class Person(Persistent):
    # …
```

Application boilerplate

- Create a storage
- Create a database object that uses the storage
- Open a connection to the database
- Get the root object (and perhaps add app collections)

```
fs = FileStorage('cal.fs')
db = DB(fs)
conn = DB.open()
root = conn.root()
if not root.has_key('collectionName'):
        root['collectionName'] = OOBTree()
        get_transaction().commit()
```

Using transactions

- After making changes
  - Get the current transaction
  - Commit or abort it

```
calendar = root['calendar']
calendar.add_appointment(app)
get_transaction().commit()
# …or…
get_transaction().abort()
```

Writing persistent classes

- Persistent if reachable from the root
- Persistency by storing/loading pickles
- ZODB must know when an object is accessed or changed
- Automatic (transparent) for attribute access
- Some common Python idioms require explicit interactions

Persistence by reachability

- Persistent object must be reachable from the root object, which ZODB creates automatically

```
person = Person(name)
people = root['people']
if not people.has_key(name):
    people[name] = person

get_transaction().commit()
```

What state is saved?

- Objects to be stored in ZODB must be picklable.
- ZODB pickles all object attributes
  - Looks in __dict__
  - Loads pickled state into __dict__
- Classes can override behavior
  - via __getstate__() and __setstate__()

References to other objects

- Sub-objects are pickled by value except:
  - Persistent sub-objects are pickled by reference
  - Classes, modules, and functions are pickled by name
  - Upon unpickling instances, __init__() is not called unless the class defined a __getinitargs__() method at pickle-time
  - See the Python 2.2 pickle module documentation for more rules regarding extension types, etc.

Automatic notice of changes

- Changes to an object via attribute access are noticed automatically by the persistence machinery
  - Implemented as tp_getattr hook in C

```
person.name = 'Barry Warsaw'
get_transaction().commit()
```

Mutable attributes

- Mutable non-Persistent sub-objects, e.g. builtin types (list, dict), instances
- Changes not caught by ZODB
  - Attribute hook only works for parent
  - Must mark parent as changed (_p_changed)

```
class Appointment(Persistent):
    # …
    def add_person(self, person):
        self.participants.append(person)
        self._p_changed = 1
```

PersistentMapping

- Persistent, near-dictionary-like semantics
  - In StandaloneZODB, inherits from UserDict
- It fiddles with _p_changed for you:

```
>>> person.contacts
<PersistentMapping instance at 81445d8>
>>> person.contacts['Barry'] = barry
>>> get_transaction().commit()
```

PersistentList

- Provides list-like semantics while taking care of _p_changed fiddling
- In StandaloneZODB only (for now)
  - Inspired by Andrew Kuchling’s SourceForge project (zodb.sf.net)
- Inherits from UserList

Handling unpicklable objects


```
class F(Persistent):
   def __init__(self, filename):
       self.fp = open(filename)
   def __getstate__(self):
       return self.fp.name
   def __setstate__(self, filename):
       self.fp = open(filename)
>>> root['files'] = F('/etc/passwd')
>>> get_transaction().commit()
>>>
```

Volatile attributes

- Attributes not to be stored persistently should be prefixed with `_v_`

```
class F(Persistent):
  def __init__(self, filename):
      self._v_fp = open(filename)
>>> root['files'] = F('/etc/passwd')
>>> get_transaction().commit()
# later…
>>> root['files'].__dict__
{}
```

Python special methods

- ExtensionClass has some limits
  - post-1.5.2 methods
  - Reversed binops, e.g. __radd__
  - Comparisons with other types
  - Ported to Python 2.2, but not being actively maintained.
- Not fundamental to approach
  - Future implementation will not use E.C.

Managing object evolution

- Methods and data can change
  - Add or delete attributes
  - Methods can also be redefined
- Classes stored by reference
  - Instances gets whatever version is imported
- __get/setstate__() can handle data
  - Provide compatibility with old pickles, or
  - Update all objects to new representation

__setstate__()


```
class Person(Persistent):
  def __init__(self, name):
      self.name = name

>>> barry = Person('Barry Warsaw')
>>> root['people']['barry'] = barry
>>> get_transaction().commit()
```

__setstate__() con’t


```
class Person(Persistent):
  def __init__(self, username,realname):
     self.username = username
     self.realname = realname
  def __setstate__(self, d):
     self.realname = name = d[‘name’]
     username = name.split()[0].lower()
     self.username = username
```

Transactions and Persistence

- This section will:
  - Explain the purpose of transactions
  - Show how to add transactions to app

Using Transactions

- ZODB adds builtin `get_transaction()`
  - Side-effect of import ZODB
- Each thread gets its own transaction
  - `get_transaction()` checks thread id
- Threads are isolated
  - Each thread should use its own DB connection
  - Changes registered with conn that loaded object
  - Synchronization occurs at transaction boundaries

ACID properties

- Atomic
  - All updates performed, or none
- Consistent
  - Responsibility of application
  - Changes should preserve object invariants
- Isolated
  - Each transaction sees consistent state
  - Transactions occur in serializable order
- Durable
  - After a commit, change will survive crash

Optimistic concurrency control

- Two alternatives to isolation
  - Locking: transaction locks object it modifies
  - Optimistic: abort transactions that conflict
- ZODB is optimistic
  - Assume conflicts are uncommon
  - If conflict occurs, abort later transaction
- Effect on programming style
  - Any operation may raise ConflictError
  - Wrap all code in try/except for this
  - Redo transaction if it fails

Transaction boundaries

- Under application control
- Transaction begin is implicit
  - Begins when object loaded or modified
- `get_transaction().commit()`
  - Make changes permanent
- `get_transaction().abort()`
  - Revert to previously committed state

Write conflicts

- Transactions must be serializable
- Two transactions change object concurrently
  - Only one change can succeed
  - Other raises ConflictError on commit()
- Handling ConflictError
  - Abort transaction, and retry
  - Application-level conflict resolution

Conflicts and Consistency

- New method on Calendar object

```
def make_appointment(self, apt, attendees):
    self.add_appointment(apt)
    for person in attendees:
        if person.is_available(apt.date, apt.duration):
            person.add_appointment(apt)
            apt.add_person(person)
```

  - Guarantees appointments don’t conflict
- Consider two calls at same time
  - Data race on is_available()?
  - Conflict raised when object commits

Conflict Example


```
def update1(cal, attendees):
    apt = Appointment("refrigerator policy",
                      Time("2/5/2002 10:00"), Time("0:30"))
    cal.make_appointment(apt, attendees)

def update2(cal, attendees):
    apt = Appointment("curly braces",
                      Time("2/5/2002 10:00"), Time("1:00"))
    cal.make_appointment(apt, attendees)
```

Two calls at once results in one error

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
File "ZODB/Transaction.py", line 233, in commit
File "ZODB/Connection.py", line 347, in commit
File "ZODB/FileStorage.py", line 634, in store
ConflictError: database conflict error (serial was
034144749675e55d, now 03414442940c1bdd)
```

Read conflicts (1)

- What if transaction never commits?
  - Operation is read-only
  - Must still have consistent view
- Always read current object revision
  - If another transaction modifies the object, the current revision is not consistent
  - ReadConflictError raised in this case
  - May need to sync() connection

Read conflicts (2)

- Example with transactions T1, T2
  - Sequence of operations
    - T1: Read O1
    - T2: Read O1, O2
    - T2: Write O1, O2
    - T2: Commit
    - T1: Read O2 – ReadConflictError
  - Can’t provide consistent view
    - T1 already saw old revision of O1
    - Can’t read new revision of O2

Multi-version concurrency control

- Planned for ZODB4
  - Allow transactions to proceed with old data
    - In previous example, T1 would see version of O2 from before T2
  - Eliminate conflicts for read-only transactions
- Limited solution exists now
  - Define _p_independent()
    - Return true if it's safe to read old revision

Example transaction wrapper


```
from ZODB.POSException import ConflictError

def wrapper(func, retry=1):
   while 1:
       try:
           func()
           get_transaction().commit()
       except ConflictError:
           if retry:
               get_transaction().abort()
               retry -= 1
               continue
       else:
           break
```

Application-level conflict resolution

- Objects can implement their own (write) conflict resolution logic
- Define _p_resolveConflict() method
  - Arguments (unpickled object states)
    - Original object state
    - Committed state for last transaction
    - State for transaction that conflicts
  - Returns new state or None or raises error
- Requires careful design
  - Can’t access other objects at resolution time
  - Must store enough info in object state to resolve

Conflicts and ZEO

- ZEO uses asyncore for I/O
  - Invalidation msgs arrive asynchronously
  - Processed when transaction commits
- Application must either
  - Start asyncore mainloop
  - Synchronize explicitly
    - Connection method sync()
    - Call when transaction begins

Subtransactions

- You can create subtransactions within a main transaction
  - individually commit and abort subtransactions
  - not “truly committed” until containing transaction is committed
- Primarily for reducing in-memory footprint

```
>>> get_transaction().commit(1)
```

Practical Considerations

- This section will:
  - Help you select components
  - Discuss sys admin issues
  - Manage resources effectively

BTrees


```
from BTrees.OOBTree import OOBTree
```

- Mapping type implemented as Btree
  - Implemented in C for performance
  - Several flavors with object or int key/values
    - OOBTree, IIBTree, OIBTree, IOBTree
- Limited memory footprint
  - Dictionary keeps everything in memory
  - BTree divided into buckets
    - Not all buckets in memory at once

Pros and cons of various storages

- FileStorage
  - Widely used (the default)
  - Large in-memory index
    - StandaloneZODB has a smaller index
  - Stores everything in one big file
- BerkeleyDB storage
  - Uses transactional BerkeleyDB
  - Large blobs (pickles) may cause performance problems
- Others:
  - OracleStorage, MappingStorage, …

Object revisions

- Each update creates new revision
  - Storages (may) keep old revisions
  - Allows application to undo changes
  - Must pack storage to remove revisions

Packing

- Some storages store multiple object revisions
  - Used for undo
  - Eventually used for multi-version concurrency control
- Old object revisions consume space
- Pack storages to reclaim
  - Can't undo
  - Experimental garbage collection

Storage management

- FileStorage
  - Packing
  - Backing up
  - Recover
- Berkeley storage
  - Packing
  - Backing up
  - Berkeley maintenance
  - Tuning

When to use ZEO

- Storages may only be opened with a single process
  - Although it may be multithreaded
- ZEO allows multiple processes to open a storage simultaneously
- Processes can be distributed over a network
- ZEO cache provides read-only data if server fails

ZEO Cache

- Disk-based cache for objects
  - Server sends invalidations on update
  - Checks validity when client connects
- Persistent caching
  - Reuse cache next time client is opened
  - Default is not persistent
- ZEO connections
  - Attempts new connection in background when server fails

ZEO management

- StorageServer
  - Run as separate process
    - May want to run under init or rc.d
  - ZEO/start.py provided as startup script
- ClientStorage

```
from ZEO.ClientStorage import ClientStorage
s = ClientStorage((“server.host”, 3400))
```

- Persistent cache

```
s = ClientStorage(host_port, client=“abc”)
```

zLOG

- Zope logging mechanism
  – More flexible than writing to stderr
- Controlled by environment vars
  – STUPID_LOG_FILE
  – STUPID_LOG_SEVERITY
- Severity levels
  – 300 to –300; 0 is default
  – -100 provides more details
  – -300 provides enormous detail

Storage migration

- Storages can be migrated via the iterator protocol
  - High level interface
  - Low level interface

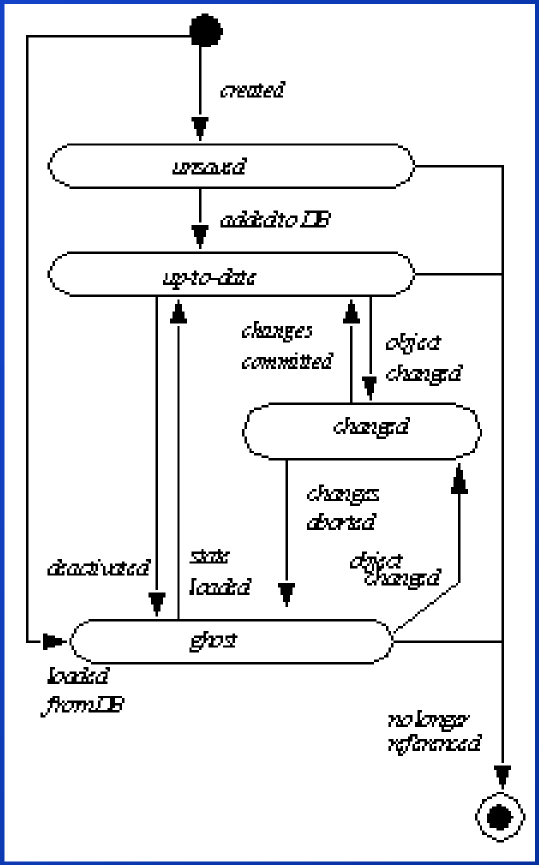
```
src = FileStorage("foo.fs")
dst = Full("BDB") # Berkeley storage
dst.copyTransactionsFrom(src)
dst.close()
```

Advanced Topics

- This section will:
  - Describe ZODB internals
  - Discuss data structures
  - Introduce various advanced features

Internals: Object state

- Objects in memory
  - Four states
    - Unsaved
    - Up-to-date
    - Changed
    - Ghost
  - Ghost is placeholder
    - No attributes loaded
    - _p_deactivate()



Internals: Connection Cache

- Each connection has an object cache
  - Objects referenced by OID
  - Objects may be ghosts
- All loads go through cache
  - Cache access to recent objects
  - Prevent multiple copies of one object
  - Creates ghost unless attribute needed
    - Note: dir() doesn’t behave correctly with ghosts

Internals: Cache Size

- Controlled by DB & Connection
- Methods on DB object
  - Affects all conns associated with DB
    - DB(…, cache_size=400, cache_deactivate_after=60)
    - setCacheSize(size)
    - setCacheDeactiveAfter(size)
    - cacheFullSweep(age)
    - cacheMinimize(age)
  - Size is objects; after/age is seconds

Advanced topics

- Undo
  - Transactional
  - destructive
    - Deprecated, but used as fallback
- Versions

Transactional Undo

- Implemented by writing a new transaction
- Supports redo
- Supported by FileStorage and BerkeleyDB storage (Full)

```
if db.supportsTransactionalUndo():
    db.undo(txn_id)
    get_transaction().commit()
```

undoLog()

- undoLog(start, stop [, func])
  - Returns a dictionary
  - Entries describe undoable transactions between start & stop time (sec. since epoch)
  - Optional func is a filter function
    - Takes a dictionary describing each txn
    - Returns true if txn matches criteria

UndoError

- ZODB.POSException.UndoError
  - raised when undo() is passed a txn id for a non-undoable transaction
  - Packing can cause transactions to be non-undoable

Versions

- Like a long running, named transaction
- If a change to an object is made in a version, all subsequent changes must occur in that version until;
  - version is committed
  - version is aborted
- Otherwise, VersionLockError

Opening a version


```
if db.supportsVersions():
   db.open(version="myversion")


# Commit some changes, then
db.commitVersion("myversion")
# … or …
db.abortVersion("myversion")
```

Database API


```
open(version="", transaction=None, temporary=0)
```

- open() returns new Connection object
  - If version specified, work in a "version"
  - If transaction specified, close connection on commit
  - If temporary specified, do not use connection pool
    - DB keeps pool of connections (and their caches) to reuse
    - Default pool size is 7
    - Locking prevents more than 7 non-temporary connections
    - Separate pools for versions
- pack(t=None, days=0)
  - Pack revisions older than t (default is now)
  - Optional days subtracts days from t