# Software Upgrades in Distributed Systems

Barbara Liskov
MIT Laboratory for Computer Science
October 23, 2001

# Examples

- Changing the algorithms and data structures in nodes making up a CFS system
- Changing a routing algorithm, e.g., Chord
- Changing the code running at some subset of nodes in an embedded system
- Changing objects in a persistent object store

# Why Upgrade?

- Upgrades are needed in long-lived systems
  - to correct implementation errors
  - to improve performance
  - to enhance behavior
  - to provide new functionality

- Note
  - must change code and data
  - not just handling a new kind of object

# Upgrade Issues

- Systems are very large
- Slow/intermittent communication
- Components might be embedded
- There may be no operator

- These are not upgrades to the code running at your PC!

# Upgrade Requirements

- Software upgrades must be propagated automatically
- Upgrade mechanism must be robust
- Limit what upgrader must do
- System must continue to run while upgrading

# Talk Outline

- Lazy upgrades in an object-oriented database
- Solving the more general problem

# Upgrades in an OODB

Object Model

- every object has a type
- objects can refer to one another and invoke one another's methods
- objects are completely encapsulated
- computations run as atomic transactions

# Examples

- Implementation of a map changes from linear to a hash table
- Circular list with one value per node now has a second value
- Sorted Set becomes Priority Set

```
void insert (Sortable x)
```

→

```
void insert (Sortable x, int x)
```

# Upgrade Requirements

An upgrade transforms the objects

- object rep might change
- object type might change
- the implementations of some methods will change

However upgraded objects must retain

- their identity and
- their state

# Base Approach

- Upgrader defines and runs an upgrade transaction
- Benefits
  - complete control of order and computation
- Drawbacks
  - writing the upgrade transaction is not easy
  - very long delay for application transactions

# Reducing Complexity

An upgrade is a set of class upgrades

<C_old, C_new, TF>

TF is the transform function

TF: C_old → C_new

System causes identity switch at some point after TF runs

# Transform Example 1

Changing map implementation

| old rep | new rep |
| --- | --- |
| `Object[ ] els;` | `HT els;` |

```
HashMap TF (LinearMap x) {
  this.els = new HT( );
  // loop over x.els and hash elements
  // into this.els
  }
```

# Transform Example 2

Adding an extra field to a circular list

| old rep | new rep |
| --- | --- |
| `CList next;` | `Clist_new next;` |
| `Object val;` | `Object val1;` |
| | `Object val2;` |

```
CList_new TF (Clist x) {
   this.next = x.next; // type-incorrect!
   this.val1 = x.val;
   this.val2 = nil;
   }
```

# Transform Function

- Transform x.next immediately
  - leads to deadlock
- Just do the assignment
  - suppose TF calls a method on this.next?

Solution:

```
CList_new TF (CList  x) {
      this.val1 = x.val;
      this.val2 = nil;
      } [next: x.next]
```

# Upgrade Completeness

Incompatible Upgrades

- C_new not a subtype of C_old, e.g.,
  - PrioritySet isn't a subtype of SortedSet

- In this case, classes that depend on the old behavior will also need to be upgraded
- Upgrade completeness can be checked
  - related to type checking

# Running an Upgrade

System determines order to apply TFs

- want same outcome for all orders
- therefore TFs must be well-behaved
  - TF must not modify any pre-existing objects
- can be lazy: objects are upgraded "just in time"
  - TF runs on x before application call x.m runs

NOTE: less expressive power than base approach

# Laziness Semantics

Separate transaction per transform

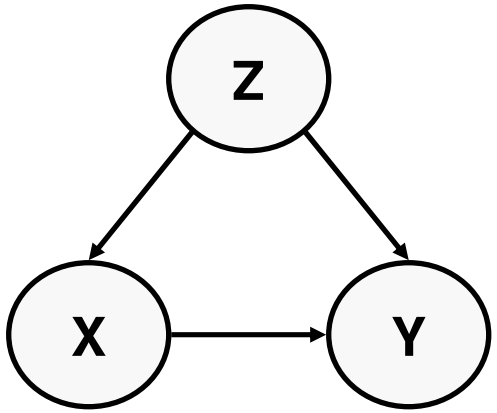
A1; A2; T3; A4; T5; ...

- Interrupt application transaction to transform x
- Commit transform transaction and switch identity: x_new takes over the identity of x
- Continue with application transaction if possible
  - will be possible if TF is well-behaved

# Laziness Justification

- Inexpensive
- Applications never notice interleaving with transform transactions

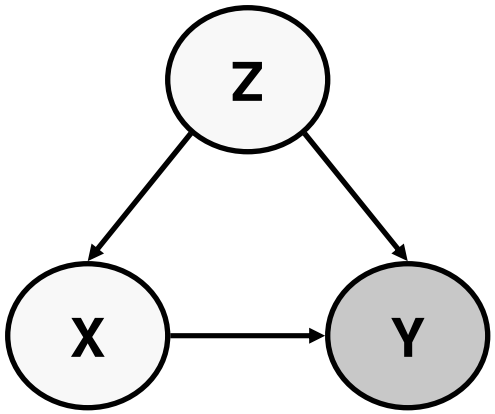
# Need Old Versions




```
z.m
    y.addEl
    x.update
```

# Need Old Versions

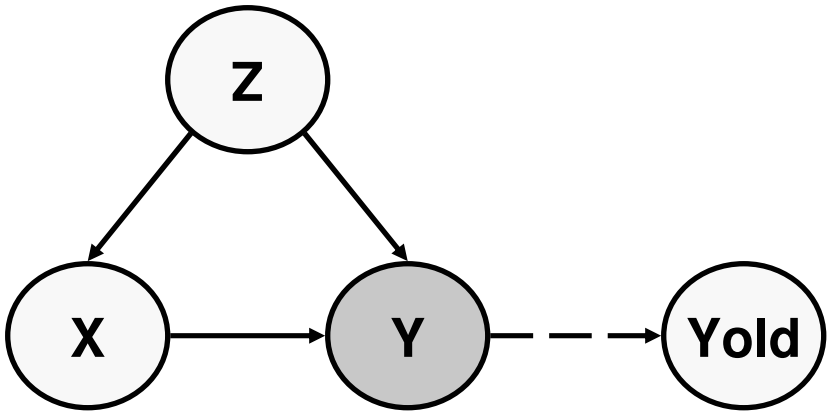
- z.m calls y.addEl; y is transformed; y.addEL runs




- z.m calls x.update; x is transformed; x.update runs

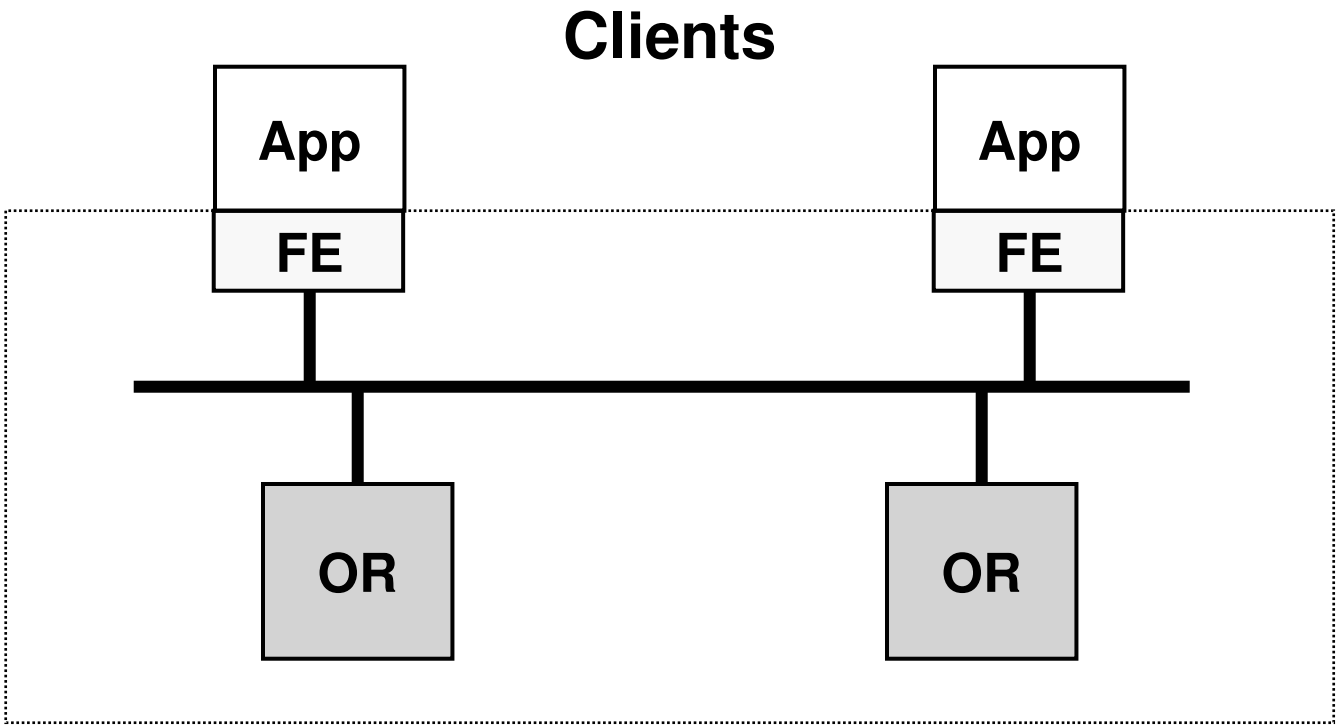
# Need Old Versions

- z.m calls y.addEl; y is transformed; y.addEL runs




- z.m calls x.update; x is transformed; x.update runs

# Implementation in Thor

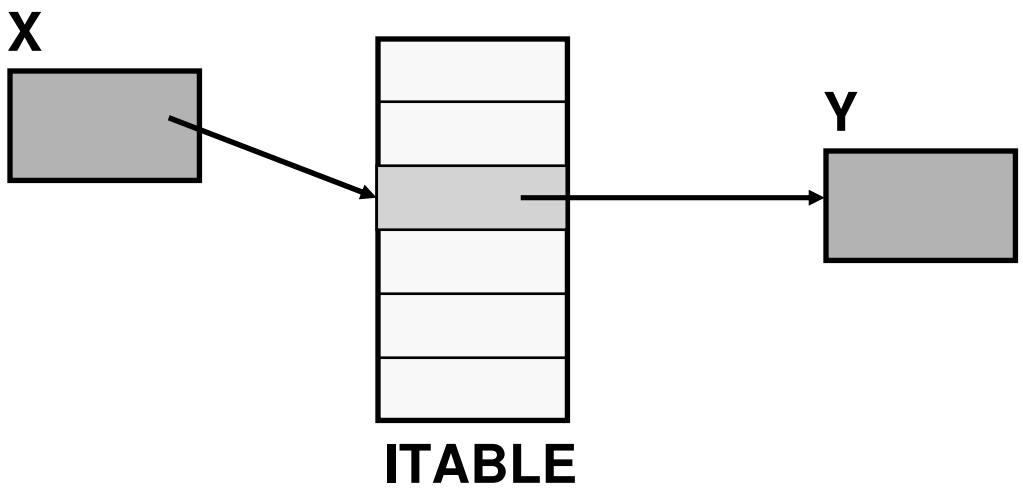


# Running Upgrades

- Defining the upgrade
  - Happens at the upgrade server (one of the ORs)
  - Upgrade server commits the upgrade if it's ok
- Propagating the upgrade
  - By gossip
- Executing the upgrade
  - FEs run the TFs
  - Could be "upgrading" FEs
  - Old versions collected by GC

# Processing at FE

- Implementation uses indirection table
- Removes old objects when upgrade arrives
  - therefore, all objects in ITABLE reflect latest upgrade



# Performance Expectation

Assumption:  upgrades are rare so optimize for non-upgrade case

- Long delay when FE first learns of upgrade
- No impact on application transactions that don't require transforms
- Otherwise delay proportional to processing of TF

# Acknowledgements


- Chandra Boyapati
- Daniel Jackson
- Liuba Shrira
- Shan Ming Woo
- Yan Zhang

# Talk Outline

- Lazy upgrades in an object-oriented database
- **Solving the more general problem**

# Upgrades in Distributed Systems

Requirements

- Automatic propagation/execution of upgrades
- Robust upgrade mechanism
- Limit what upgrader must do
- System must continue to run while being upgraded

- Upgrade may take effect slowly, e.g., disconnected nodes, slow links, controls
- Nodes running different versions may need to communicate

# Insight/Hypothesis

Robust systems can be upgraded

- They survive node restarts
- They provide service even when some nodes are down
- A node can do its job even when it can't communicate with some other nodes

Therefore, upgrade can be a (soft) restart

# Upgrade Model

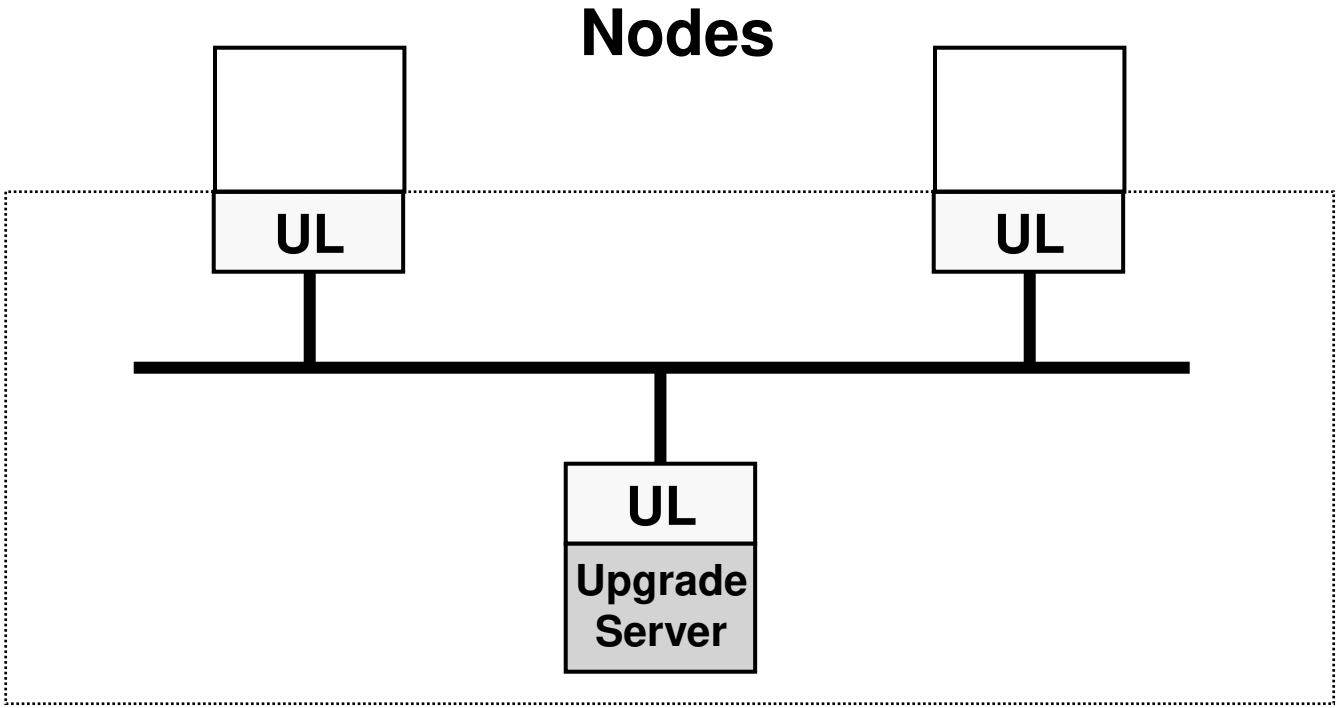
- Each node is an object
  - it retains its identity and its state
- Node upgrade involves running TF
- Node upgrade is atomic
- But upgrade might be lazy within a node
  - running the TF can take time!

# Examples

- Thor has ORs and FEs
  - FEs provide client interface
  - ORs have two interfaces (to ORs, to FEs)
  - protocols using TCP/IP

- Example upgrades
  - change FE implementation
  - FE/OR protocol changes (e.g., invalidations)
  - OR/OR protocol changes (e.g., commit protocol, GC)

# System Architecture




- UL is the Upgrade Layer
  - all messages go through it (lightweight)
  - plus its own protocols

# Step 1: Defining Upgrades

- Happens at upgrade server
- Issues
  - Who can do it?
  - Correctness checking, e.g., completeness, correctness of TF
  - Control of scheduling
  - Defines ordering (version number)

- Undoing an upgrade?
- Monitoring an upgrade?

# Step 2: Propagating Upgrades

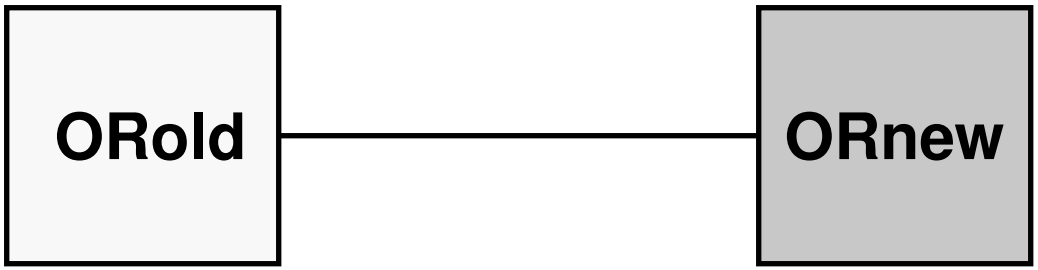
- Done by the upgrade layer
- Base mechanism: check with upgrade server periodically
  - uses upgrade layer protocol
- Gossip: piggyback on node communication
  - because upgrade layer processes every message

- Upgrade layer communicates with the upgrade server

# Step 3: Executing an Upgrade

- Done by upgrade layer
- Decides when to run the upgrade
  - Upgrade runs after it arrives
- Shuts the node down (soft)
- Fetches new code
- Runs the TF
  - may require communication (implies multi-versions)
  - may be lazy
- Restarts the node

# Running in a “mixed” System




Problems only when node interface or external behavior changes

# Failure Model for Upgrades

The upgrade layer

- Rejects incoming calls to old unsupported methods, e.g., from ORold to ORnew
- Treats outgoing calls of unhandled new methods as node failures, e.g., from ORnew to ORold

Disadvantage: upgrades may need to be installed quickly

# Simulation Model for Upgrades

The upgrade layer

- handles all old incoming calls, e.g., from ORold to ORnew
  - upgrades must be backward compatible
  - but can deprecate methods
- simulates outgoing calls of new methods if necessary, e.g., from ORnew to ORold

Disadvantage:  more complex

- upgrader must supply a proxy to handle incoming and outgoing calls at the upgraded node

# Comparison

- Upgrades are similar in OODBs and in distributed systems
  - Both define TFs on “classes”
  - Completeness matters in both
  - TF runs as a transaction interleaved with applications
  - Still need old versions to support running TF
- But they are also different
  - Now application might run before TF

# Summary

Upgrades in an OODB

- can be lazy
- takes advantage of transactions
- introduces concepts with wider application (transform functions, completeness)

Upgrades in a distributed system

- robust systems can be upgraded
- they are transactional in some sense
- needs an upgrade layer/architecture

# Future Work

Upgrades in distributed systems!

- failure or simulation model for upgrades
- controlling scheduling of upgrades
- lazy TF
- node is more than one object
- downgrades

# Software Upgrades in Distributed Systems

Barbara Liskov
MIT Laboratory for Computer Science
October 23, 2001