Software Upgrades for Distributed Systems

Sameer Ajmani
Google

Barbara Liskov
MIT

Liuba Shrira
Brandeis

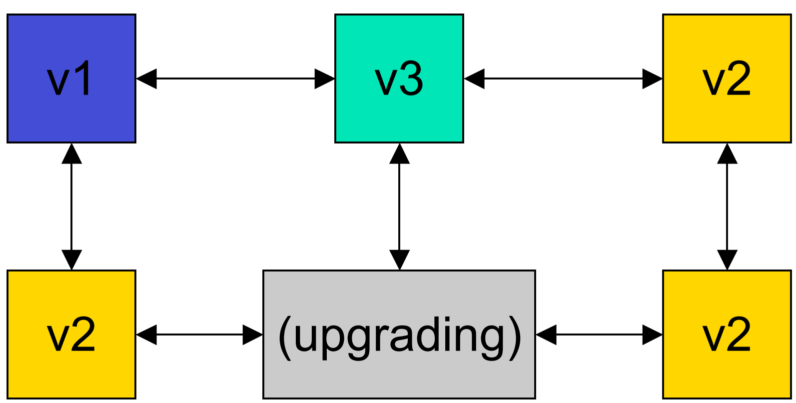
Internet Services

- Are long-lived, robust
- Run on many machines
- Must be continuously available
- Have persistent state
- Face ever-changing requirements

- Require software upgrades to
  - Fix bugs
  - Improve performance
  - Add/change/remove features

Upgrade Requirements

- Automatic, Controlled Deployment
  - Ensure continuous availability
  - Test new software on a few nodes
  - Upgrade servers before clients
- **Mixed mode operation**



Outline

- System & Upgrade Model
- Specifying Upgrades
- Implementation Models

System Model

- A node is an object of class C
- Different nodes may run different classes
- Nodes communicate via RPCs

Upgrade Model

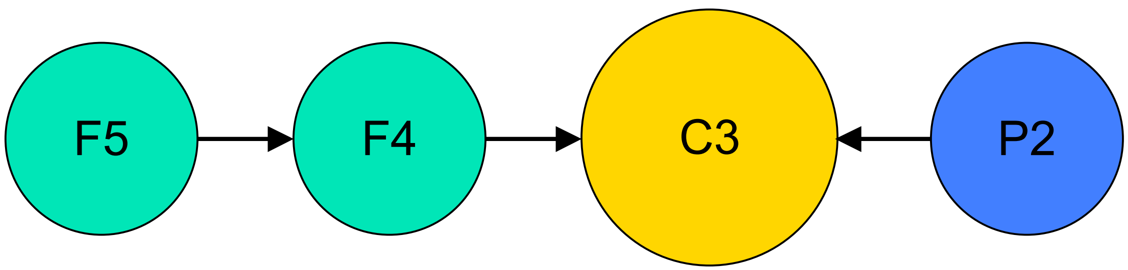
- A class upgrade replaces an old class, C_{old}, with a new one, C_{new}
  - Implements types T_{old}, T_{new}
  - May be compatible or incompatible
- An upgrade is a set of class upgrades

Supporting Mixed Mode

- Each node handles calls to past and future versions of itself
- Adding support for new versions must be fast
- Removing support for old versions should be easy

Simulation Objects

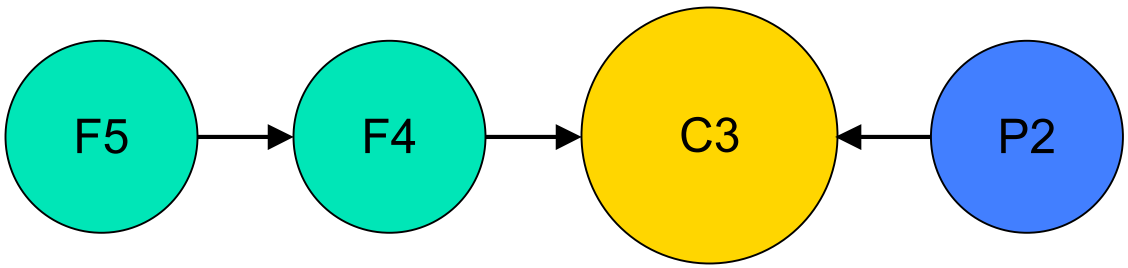
- Each node handles calls to past and future versions of itself



- Future SOs simulate future behavior
- Past SOs simulate past behavior
- Adding/removing an SO does not require a restart

Simulation Objects

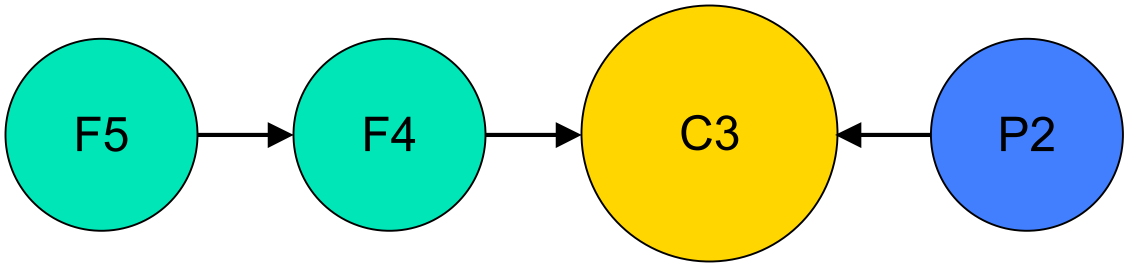
- Each node handles calls to past and future versions of itself



- SOs are only required for certain upgrades

Specifying Upgrades

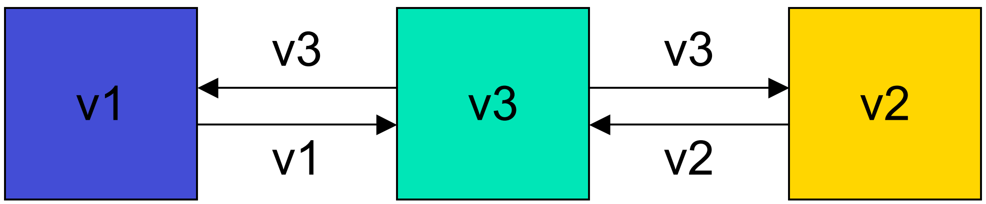
Specifying Upgrades



- Must behave like a single object
  - Even when upgrades are incompatible
- Upgrade specification must define this
  - Goal: no surprises for clients!
  - E.g., changing permissions to ACLs

Constraints on Specifications

- Type requirement
  - A call of version V behaves according to the specification of type T_V



Constraints on Specifications

- Sequence requirement
  - Each event must reflect all earlier ones despite:
    - Client upgrades
    - Server upgrades
    - Version introduced
    - Version retired

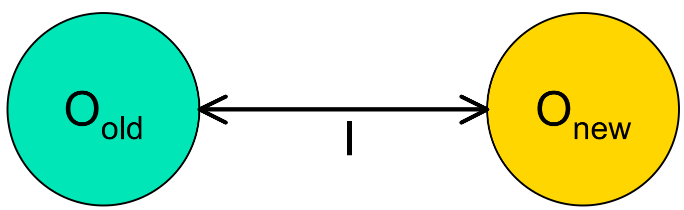
Example

- ColorSet ◊ FlavorSet
  - Incompatible upgrade
- ColorSet methods: insertColor(x, c), getColor(x), …
  - E.g., { (1, red), (2, blue), (3, red) }
- FlavorSet methods: insertFlavor(x,f), getFlavor(x), …

Specifications: Invariant

Invariant I relates the object states

$I(O_{old}, O_{new})$

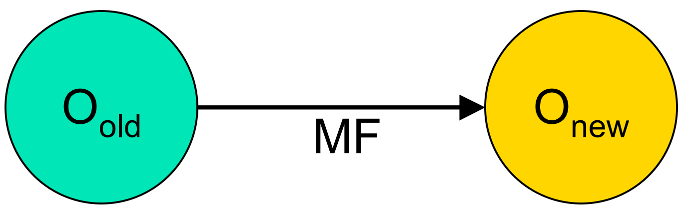


$I\text{:}\ \ \{x | \langle x,c \rangle \text{ in } O_{CS}\} = \{x | \langle x,f \rangle \text{ in } O_{FS}\}$

Specifications: Mapping Function

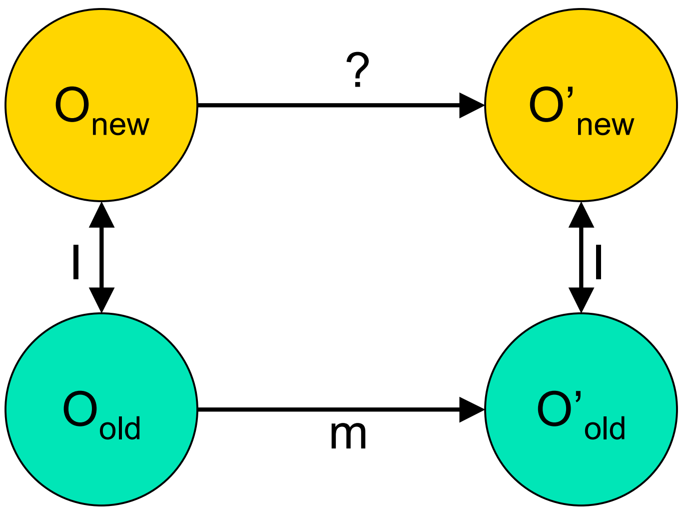
Mapping function MF defines initial state

$O_{new} = MF(O_{old})$ s.t. $I(O_{old}, O_{new})$



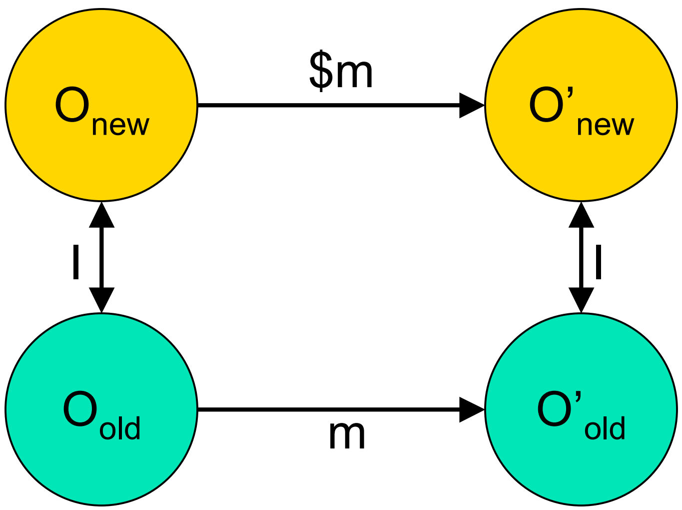
$O_{FS} = MF(O_{CS}) = \{<x,grape>|\ <x,c> \text{ in } O_{CS}\}$

Relating Behavior



Only for mutators

Relating Behavior



Shadow methods relate behavior

Told.m ◊ Tnew.$m

Tnew.p ◊Told.$p

Shadow Method Specification

void ColorSet.$insertFlavor(x, f)

Effects: no <x,c> in $\text{this}_{pre} \Rightarrow$

$\text{this}_{post} = \text{this}_{pre}$ U {<x,blue>}

- Also ColorSet.$delete, FlavorSet.$insertColor, FlavorSet.$delete

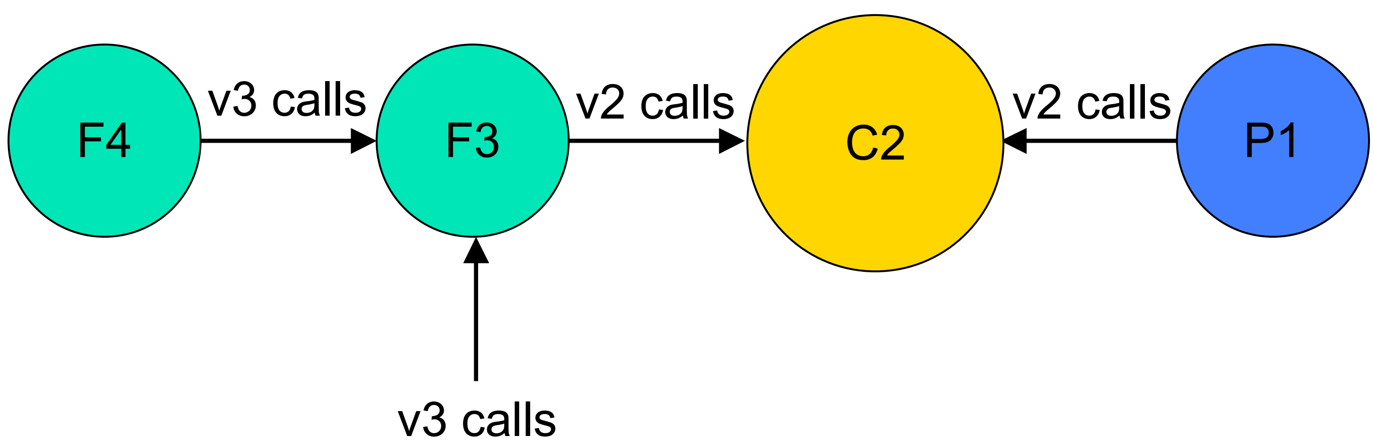
The Compound Type

- I, MF, and the shadow methods define a compound type $T_{old\&new}$
  - All the methods, with extended specs for mutators
- We would like:
  - $T_{old\&new}$ is a subtype of T_{old}, T_{new}
- When this doesn't work:
  - Weaken invariant I
  - Upgrade scheduling
  - Disallow methods

Implementation Models

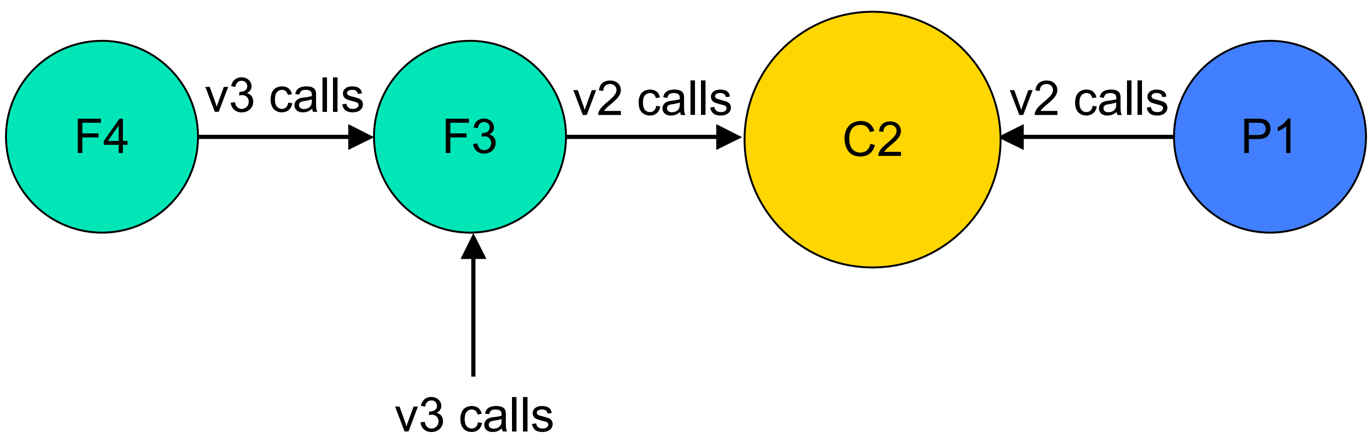
Direct Model

- SO handles calls only to its own version
  - And delegates down the chain



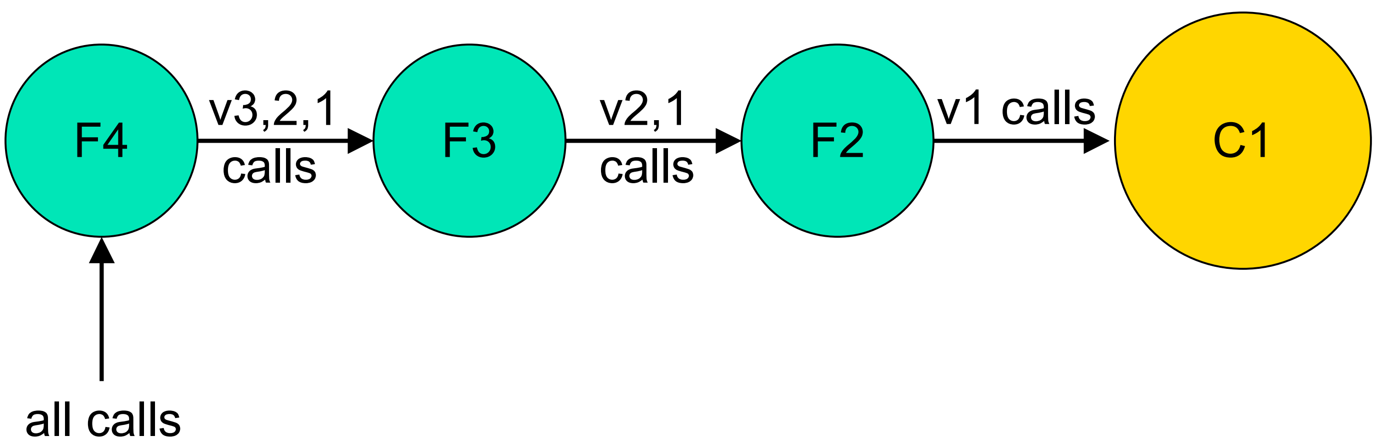
Problems with Direct Model

- Poor expressive power
  - E.g., FlavorSet SO doesn't know about C.insertColor call
- Synchronization



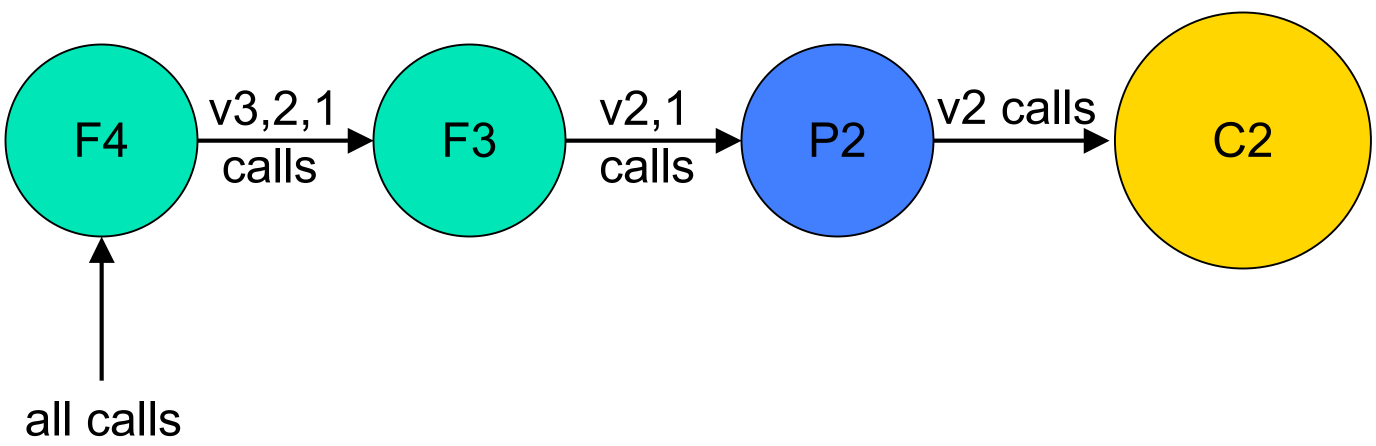
Interceptor Model

- Newest SO gets all calls
  - And delegates down the chain



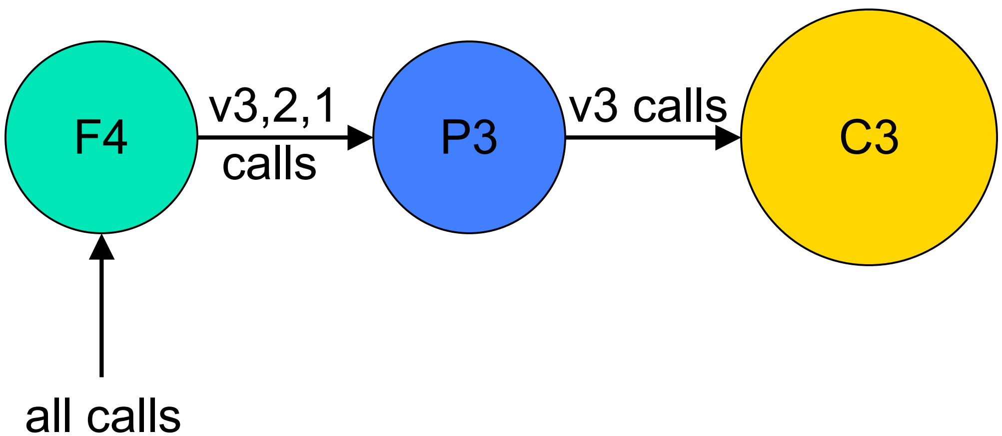
Interceptor Model

- After first (incompatible) upgrade is installed



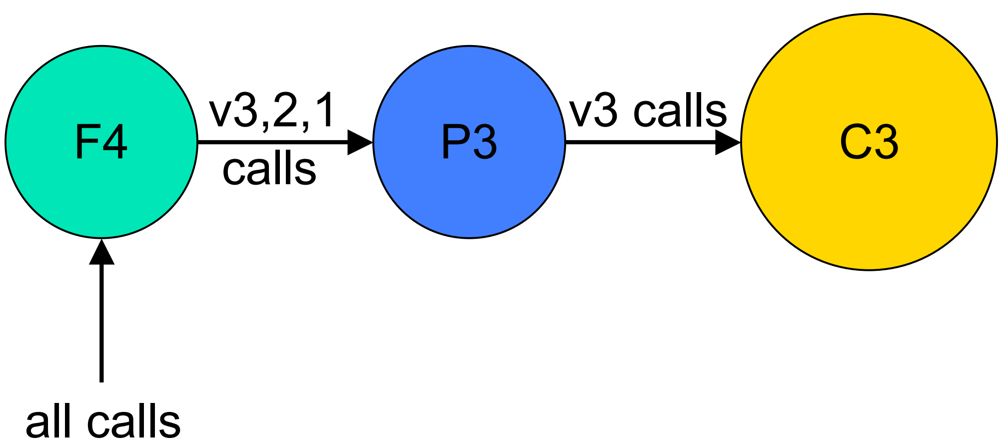
Interceptor Model

- After second (possibly compatible) upgrade is installed



Interceptor Model Evaluation

- Excellent expressive power
- Future and past SO must do more
  - Can reuse code and delegate



Prototype Implementation: Upstart

- C++ and Sun RPC
- Intercepts socket(), read(), write()
- Imposes minimal overhead

Summary

- Upstart is the first complete approach
  - Allows mixed mode operation
- The first definition of what must be specified for incompatible upgrades
- A powerful and useful implementation model
- A prototype implementation

Software Upgrades for Distributed Systems

Sameer Ajmani

Google

Barbara Liskov

MIT

Liuba Shrira

Brandeis

Disallowing

- Constraint: never disallow methods of the current object
- Future SO may disallow T_{new} methods
- Past SO may disallow T_{old} methods
- In either case, disallow
  - Mutators whose shadows are problem
  - Observers that expose problems

Some shadows cannot be implemented via delegation

- Disallow methods that have unimplementable shadows
- Add shadow method to delegate via dynamic updating
  - Allowed iff T + shadow is a subtype of T
  - E.g., can’t add delete() to GrowSet
- Implement shadow method in interceptor
  - Impacts transform function
  - Won’t work for past SOs because of retirement

What does Google do?

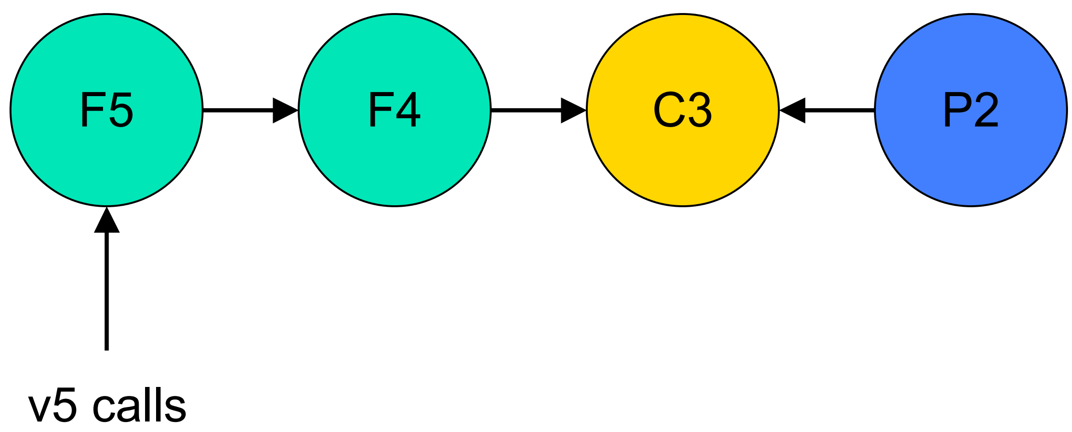
- Extensible protocols
  - Assume defaults for missing fields
  - Ignore unexpected fields
- Round-robin upgrades among replicas
- Datacenter-by-datacenter

END OF SLIDES

- The remaining slides are leftover from previous talks and may contain stale information

Code Execution

- Call contains version number
- Called node dispatches



Upstart

- A system that supports upgrades
- And a methodology

- Joint work with
  - Barbara Liskov
  - Liuba Shrira

Class Upgrade

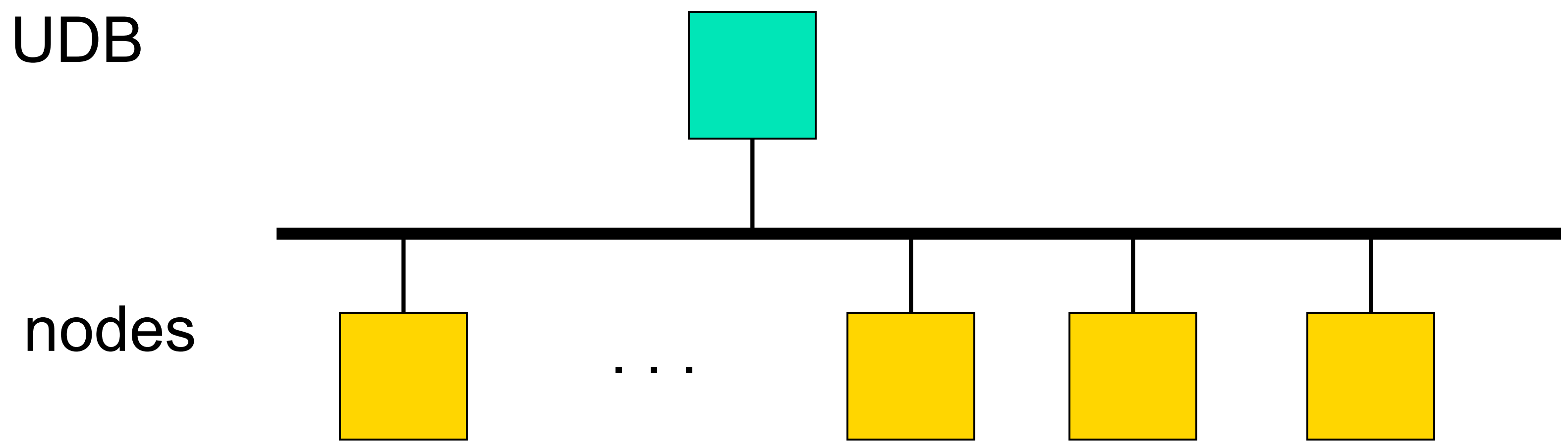
- New and old classes C_{new}, C_{old}
  - Implement T_{new}, T_{old}
- Scheduling function SF
- Transform function TF
- Simulation classes S_{new}, S_{old}

- Might be incompatible
  - T_{new} is not a subtype of T_{old}

Class Upgrade

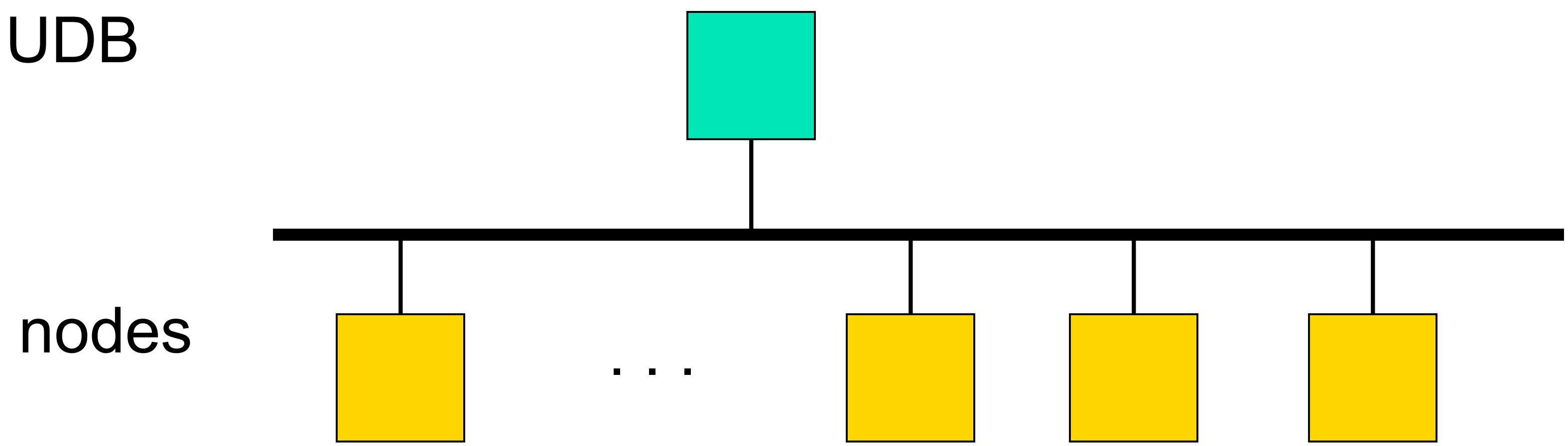
- Replaces an old class, C_{old} with a new one, C_{new}
- Every node running old class will switch to new class eventually
- Upgrade is a set of class upgrades

Defining an Upgrade



- Upgrader enters new upgrade at UDB
- Defines a new version

Propagating an Upgrade



- Nodes query the UDB periodically
- Version numbers flow on all messages

Executing an Upgrade

- If upgrade affects the node
  - Runs the SF
    - And simulates the future
  - Shuts down, restarts, runs TF
  - Starts up “normally”
    - And simulates the past

Disallowing Example

- GrowSet ◊ IntSet
- For the future SO:
  - Disallow IntSet.delete
- For the past SO:
  - Disallow GrowSet.isIn

- $T_{old\&new}$ becomes $<T_{future}\ ,\ T_{past}>$

What Disallowing Provides

- T_{future} is a subtype of T_{old}
- And it implements T_{new}

- T_{past} is a subtype of T_{new}
- And it implements T_{old}

Transform Functions

- Implement the identity map
- May need to use future SO, create past SO
- Must be restartable
- Cannot make remote calls

Scheduling Functions

- Can consult the UDB
- Examples:
  - Rolling upgrade
  - Big flip
  - Fast reboot

Implementing Upgrades

- Need to provide SOs, TF, SF
- For the SOs, need an implementation model

Summary of Specifications

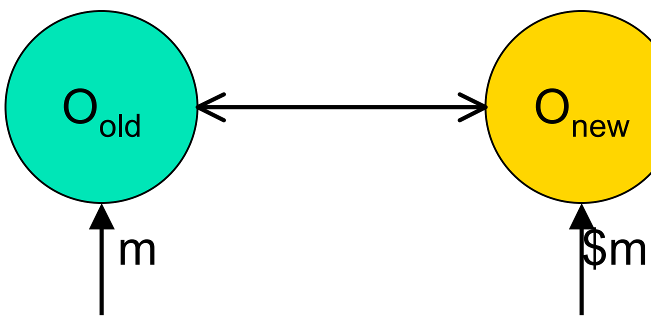
- Specification defines the compound type $T_{old\&new}$
  - I, MF, and the shadows
- If the compound type isn't a subtype, disallow

Specifications: Shadow Methods

- Shadow methods relate behavior

  $T_{old}.m \lozenge T_{new}.\$m$

  $T_{new}.p \lozenge T_{old}.\$p$



e.g., FlavorSet.$insertColor

Talk Outline

- Upgrade requirements
- Upstart overview
- Specifying upgrades
- Implementing upgrades

Requirement: Generality

- Support for arbitrary changes
- Incompatible upgrades
  - Old features are no longer supported

Requirement: Continuous Availability

- Service is required 24/7
- Even when upgrading
- Therefore systems upgrade gradually

- Implies mixed-mode operation

Requirement: Controlled Deployment

- Systems upgrade gradually
  - But with control
- Manual control is impractical
  - An automatic system
  - But upgrader needs control

Requirement: Persistence

- Systems store important state for users
  - It cannot be lost
  - But may need to be transformed

Requirement: Ease of Use

- Avoiding feature creep helps
- Upgrader needs to understand only a few recent versions