Automatic Software Upgrades for Distributed Systems

by

Sameer Ajmani

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

Author . . . .
Department of Electrical Engineering and Computer Science
August 18, 2004

Certified by . . . .
Barbara H. Liskov
Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Automatic Software Upgrades for Distributed Systems

by

Sameer Ajmani

Submitted to the Department of Electrical Engineering and Computer Science
on August 18, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Upgrading the software of long-lived, highly-available distributed systems is difficult. It is not possible to upgrade all the nodes in a system at once, since some nodes may be unavailable and halting the system for an upgrade is unacceptable. Instead, upgrades may happen gradually, and there may be long periods of time when different nodes are running different software versions and need to communicate using incompatible protocols. We present a methodology and infrastructure that address these challenges and make it possible to upgrade distributed systems automatically while limiting service disruption.

Our methodology defines how to enable nodes to interoperate across versions, how to preserve the state of a system across upgrades, and how to schedule an upgrade so as to limit service disruption. The approach is modular: defining an upgrade requires understanding only the new software and the version it replaces.

The upgrade infrastructure is a generic platform for distributing and installing software while enabling nodes to interoperate across versions. The infrastructure requires no access to the system source code and is transparent: node software is unaware that different versions even exist. We have implemented a prototype of the infrastructure called Upstart that intercepts socket communication using a dynamically-linked C++ library. Experiments show that Upstart has low overhead and works well for both local-area and Internet systems.

Thesis Supervisor: Barbara H. Liskov
Title: Professor

Acknowledgments

First, I thank my advisor, Barbara Liskov, for insisting that I understand my research as deeply as possible and present it as clearly as possible. I have learned a lot working with her.

Next, I thank my thesis readers: Michael Ernst, Daniel Jackson, and Liuba Shrira. I thank Michael for his encouragement, support, and good advice. I thank Daniel for forcing me to consider several important details that I might have otherwise overlooked. I thank Liuba for dissenting with Barbara and me often in the course of this research: most of the time, we eventually realized that Liuba was right all along.

I thank all my (past and present) coworkers here at MIT for providing good friendship and fun times: Atul Adya, Sarah Ahmeds, Chandrasekhar Boyapati, Miguel Castro, Kathryn Chen, Dorothy Curtis, Anjali Gupta, Ben Leong, Nick Mathewson, Chuang-Hue Moh, Andrew Myers, Steven Richman, Rodrigo Rodrigues, Ziqiang Tang, Ben Vandiver, Sean-Ming Woo, and Yan Zhang.

I thank my parents for their lifetime of support. Dad set a great example of how to succeed with hard work. Mom made sure I never got so caught up in work that I lost track of what really matters.

I thank my hyper-talented sister, Shama, a.k.a. Angel the Fire Artist, for providing much-needed support during difficult times and for just being so darn cool.

Finally, I thank my wife, Mandi. I cannot put into words what her love and support have done for me. All I can say is that without her, I wouldn't be where I am today. She is the greatest blessing of my life, and I thank God every day for bringing us together.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Long-lived Internet services face challenging and ever-changing requirements. Services must manage huge quantities of valuable data and must make that data available continuously to rapidly growing client populations. Examples include online email services [19], search engines [34], persistent online games [11], scientific and financial data processing systems [6], content distribution networks [14], and file sharing networks [8, 15].

The systems that provide these services are large: they are composed of hundreds or thousands of machines, and machines in different data centers are separated by the untrusted and unreliable Internet. At such scales, failure is the norm: some fraction of machines will suffer from hardware and software failures; the network will drop messages and sometimes partition altogether; and the occasional malicious attack or operator error can cause unpredictable and catastrophic faults.

As a result, the software for these systems is complex and will need changes (upgrades) over time to fix bugs, add features, and improve performance. The fundamental challenge addressed in this thesis is how to upgrade the software of long-lived distributed systems while allowing those systems to continue to provide service during upgrades.

1.1 Goals

Our aim is to create a flexible and generic *automatic upgrade system* that enables systems to provide service during upgrades. This section describes the goals for the upgrade system; in this context, the *upgrader* is the person who defines upgrades and uses the upgrade system.

The first set of goals has to do with the *upgrade model*, i.e., what an upgrade can be:

Simplicity The upgrade model must be easy to use. In particular, we want *modularity*: to define a new upgrade, the upgrader should only need to understand the relationship between the current version of the system software and the new one.

Generality The upgrade model must not limit expressive power, i.e., an upgrade should be able to change the software of the system in arbitrary ways. This goal has two parts:

Incompatibility The new version must be allowed to be incompatible with the old one, e.g., it can stop supporting legacy behavior and can change communication protocols. This is important because otherwise later versions of the system must continue to support legacy behavior, which complicates software and makes it less robust.

Persistence The systems of interest have valuable persistent state that must survive upgrades. Therefore, upgrades must allow the preservation and transformation of persistent state. This can be costly, because each node may have very large state (e.g., gigabytes or terabytes), and transforming this state to the representation required by the new software may require reading and writing the entire state (e.g., to add a new property to every file in a file system). Even modest transforms are time-consuming: the maximum read throughput of the fastest enterprise-class disks today is around 150 MB/s, and the read/write throughput of most disks is between 10 and 40 MB/s [103]. A transform that reads then writes 10 GB of data may require from two to 30 minutes of downtime; and a transform of 100GB may require five hours of downtime.

It is explicitly not a requirement that upgrades preserve volatile state. Upgrades are not very frequent, because organizations do not want to release new software until, ideally, it is free of errors; in reality they aim to get rid of most errors through rigorous testing. Therefore, new versions are deployed on a schedule that includes time for development and testing. Such a schedule might allow three months for a minor release and significantly more time for a major release. Organizations might release patches more frequently, but even then it is likely to take a few days or weeks before the new software is ready to go. Therefore, it is acceptable to lose volatile state in such infrequent events; but it is not acceptable to lose persistent state, as there may be no way to recover it. For example, it is acceptable to drop open connections and uncommitted writes to a file system, but it is not acceptable to lose the files themselves.

The second set of goals has to do with how an upgrade happens:

Automatic Deployment The systems of interest are too large to upgrade manually (e.g., via remote login). Therefore, upgrades must be deployed automatically: the upgrader defines an upgrade at a central location, and the upgrade system propagates the upgrade and installs it on each node.

Controlled Deployment The upgrader must be able to control when nodes upgrade with the same precision as if the upgrader did it manually. There are many reasons for controlled deployment, including: enabling a system to provide service while an upgrade is happening, e.g., by upgrading replicas in a replicated system one-at-a-time (this is especially important when the upgrade involves a time-consuming state transform); testing an upgrade on a few nodes before installing it everywhere; and scheduling an upgrade to happen at times when the load on the nodes being upgraded is light.

Mixed Mode Operation Controlled deployment implies upgrades are *asynchronous*, i.e., nodes can upgrade independently and at any time. This means there may be long periods of time when the system is running in *mixed mode*, i.e., when some nodes have upgraded and others have not. Nonetheless, the system must provide service, even when the upgrade is incompatible. This implies the upgrade system must provide a way for nodes running different versions to interoperate (without restricting the kinds of changes an upgrade can make).

1.2 System Model

We are interested in providing upgrades for large, long-lived distributed systems. For our purposes, a distributed system is any collection of nodes (machines) that cooperate to perform a task. Nodes are connected by a network and coordinate their actions by exchanging messages. We assume an asynchronous, unreliable network that may delay, lose, reorder, duplicate, or modify messages. Links may go down; nodes may disconnect and continue to run; and the network may partition for extended periods of time.

Our approach takes advantage of the fact that long-lived systems are *robust*. These systems tolerate communication problems: remote procedure calls may fail, and callers know how to compensate for such failures, e.g., by degrading service or retrying with another node.

Robust systems are are prepared for nodes to fail at arbitrary times. Nodes can recover from failure; when they do, they restart their software and rejoin the system. Nodes also recover their persistent state, e.g., they store it on disk, and when they recover, they initialize their state from what is on disk.

We will take advantage of our robustness assumption to upgrade nodes: a node upgrades by failing (and losing its volatile state), replacing the old software with the new software, transforming its persistent state (if required), and restarting with the new software (which recovers from the newly-transformed state).

1.3 Our Approach

To create an upgrade, the upgrader defines the new software for the system and some additional information to support the controlled deployment, persistence, and mixed-mode requirements. The upgrader then "launches" the upgrade, and the upgrade system does the rest.

The additional information consists of the following software components:

Scheduling Functions define when nodes should upgrade. We provide support for a wide variety of schedules.

Transform Functions define how nodes' persistent state must change as required by the new version. They can be omitted if no change is required.

Simulation Objects are adapters [50] that enable nodes to support calls from nodes running other versions. They are only needed for upgrades that change communication protocols. There are two kinds:

- **Future Simulation Objects** define how nodes handle messages intended for their new software before they upgrade.
- **Past Simulation Objects** define how nodes handle messages intended for their old software after they upgrade.

Because upgrades are infrequent, we expect that common case is that nodes run the same software version. We optimize for this case by enabling such nodes to communicate efficiently. Our approach will work correctly even when upgrades occur frequently and cross-version communication is common, but system performance may degrade.

An important feature of our approach is that it separates the responsibilities of the implementor of the system software and the upgrader. The implementor creates new software for the system and ensures that it works when all the nodes are running just the new software. The upgrader defines the upgrade itself, which includes defining the components described above.

Separating the responsibilities of the implementor and the upgrader makes software development easier. Many systems nowadays are burdened with providing support for legacy behavior. This makes software more complex, more difficult to maintain, and less robust. Moving support for interoperation out of the software and into simulation objects can make systems simpler and more robust. And since simulation objects are separate modules from the system software, using them does not restrict how the system software is implemented.

This separation of responsibilities also means the upgrader does not need to understand the details of the system software implementation and does need access to its source code. The upgrader just needs to understand the interfaces between nodes and how they structure their persistent state.

Our approach gives the upgrader lots of flexibility. Some upgrades can be done eagerly, i.e., nodes upgrade as soon as they learn that a new version is available. This is appropriate if the upgrade fixes a major error or if the disruption caused by installing the new software is minor. Other upgrades can be done more slowly, in a way that allows further testing or that allows clients to move gradually to a newer version. Using simulation objects to enable clients to "run in the past" reduces pressure on an organization to make new versions backward compatible and so enables them create simpler (and more likely correct) software.

Once the upgrader has defined an upgrade, the *upgrade infrastructure* takes over and makes the upgrade happen automatically. The upgrade infrastructure consists of a central *upgrade server* that stores upgrade definitions and per-node *upgrade layers* that propagate and install upgrades. Upgrade layers also enable the system to support mixed mode by intercepting all inter-node communication and using simulation objects to translate between nodes running different versions. Perfect simulation is not always possible; when it's not, some cross-version calls may fail and service may degrade.

Chapter 2 presents the details of how an upgrade happens. Chapters 3 and 4 discuss when cross-version calls may need to fail.

1.4 Contributions

We make two major contributions: a methodology for defining automatic upgrades and an infrastructure for deploying them.

Our methodology includes new techniques for scheduling upgrades (scheduling functions), managing persistent state (transform functions), and enabling cross-version interoperation (simulation objects). The methodology allows for exceptionally expressive simulation objects and can enable interoperation between nodes separated by arbitrary numbers of upgrades. Nonetheless, the methodology is modular: to define a new upgrade, the upgrader only needs to understand the relationship between the current version of the system software and the new one.

A vital part of our methodology is a new way to specify multiple types for a single node and maintain relationships between the states accessible via those types. This enables clients to know what to expect when they upgrade and start using the system via a new version or when they interact with other clients running different versions. The methodology also introduces several models for how to use simulation objects to implement a node's types.

The second major contribution of this thesis is a new infrastructure for automatically deploying upgrades on distributed systems. The design of the infrastructure is generic: it can be realized for a variety of platforms, from distributed object systems [4, 60, 79] to systems whose processes communicate via raw sockets.

We have implemented a prototype upgrade infrastructure called Upstart. Upstart intercepts communication at the socket layer using a dynamically-linked C++ library and so is transparent to applications. We have measured the overhead of Upstart for several applications on both local-area networks and on the Internet, and we show it to be practical for many kinds of systems. We have also run large-scale upgrade experiments on PlanetLab to demonstrate that our approach scales and works well.

1.5 Related Work

We review related work briefly here; we provide a full discussion in Chapter 9.

There are many real-world systems that enable an administrator to manage the software of nodes in a distributed system from a central location [1, 3, 5, 9, 10, 18, 55, 96, 104]. Unlike our approach, these do little to ensure that a system continues to provide service during an upgrade. The fundamental problem is that these approaches do not enable a system to provide service in mixed mode, i.e., when some nodes have upgraded and others have not.

Real-world organizations typically avoid mixed mode by installing upgrades during scheduled downtime, but this approach is unacceptable for systems that must provide continuous service.

For systems composed of several independent (non-communicating) data centers, it is possible to provide service during upgrades by upgrading one data center at a time and redirecting clients to the non-upgrading data centers [51]. This approach requires vast resources and does not work when nodes in different data centers must communicate.

When mixed mode cannot be avoided, systems must somehow enable upgraded and non-upgraded nodes to interoperate. Many real-world organizations do this by restricting how an upgrade may change the system software. For example, if an upgrade cannot change how nodes communicate, upgraded and non-upgraded nodes can always interoperate. This requirement can be relaxed for client-server systems: an upgrade can change the client-server protocol provided the change is *backward compatible*, i.e., the new protocol allows non-upgraded clients to work correctly with upgraded servers. But this approach does not work for server-to-server systems, because upgraded servers cannot necessarily work correctly with non-upgraded ones.

To support mixed mode in server-to-server systems, Google [51], Gnutella [15], and the Internet web [82] and mail [41] standards use *extensible protocols*. In these systems, all nodes support a common baseline protocol, and upgrades define extensions to the baseline protocol. Nodes ignore extensions they do not understand, so upgraded nodes must be able to provide service with or without the extensions. The problem with this approach is that it complicates the system software and does not support changes to the baseline protocol.

Research approaches to upgrading distributed systems generally avoid mixed mode by upgrading all the nodes that need to upgrade at the same (real or logical) time. "Reconfigurable" distributed systems [21, 28, 59, 66, 90] enforce synchrony by quiescing the nodes that need to upgrade. "Transactional" approaches [29, 102] serialize upgrades in the sequence of operations of a system, i.e., they prepare the new software on the nodes that need to upgrade, then cause those nodes do an *atomic switchover* to the new software. All these approaches stall when nodes are unavailable or when there are communication problems, and each approach is specific to a particular distributed object system.

Previous approaches try to minimize disruption by waiting until nodes quiesce [21,28,59,66,90] or reach pre-defined reconfiguration points [29, 46, 48, 54, 57, 59, 102] before upgrading them; and while a node upgrades, calls to it are queued by the system. These approaches assume transform functions are fast. We decided on our approach because it's much simpler for the person defining the upgrade (and therefore more likely that the node upgrade will execute correctly), it allows clients to retry calls that fail because a node is upgrading (rather than waiting on blocked calls), and it allows for lengthy transforms.

Our approach to state transformation is a departure from previous approaches [29, 48, 59] that attempt to preserve a node's volatile state. These approaches require that the software implementor provide routines to export a node's volatile state to / import it from a canonical representation [56]. Since we assume volatile state may be lost at any time due to node failure, it makes little sense to complicate upgrades attempting to preserve it. Instead, we keep things simple: transform functions operate only on persistent state, so no import or export routines are necessary.

The idea of using adapters [50] to enable interoperation is not new. This approach arises not only in upgradeable distributed systems [93, 102], but also in object-oriented databases [81, 97], procedure-wise upgradeable programs [48], and federated distributed systems [45, 78, 88, 94]. What distinguishes our approach is the exceptional expressive power of simulation objects and the criteria we provide for reasoning about their correctness.

1.6 Outline

The thesis is organized as follows. Chapter 2 presents our model for automatic upgrades, and Chapter 3 describes how to specify them. Chapters 4, 5, and 6 discuss the three core components of automatic upgrades: simulation objects, transform functions, and scheduling functions, respectively. Chapter 7 describes Upstart, our prototype implementation of the upgrade infrastructure, and Chapter 8 evaluates its overhead on several applications. Chapter 9 discusses related work, and Chapter 10 concludes.

Chapter 2

Upgrades

This chapter presents an overview of our methodology and infrastructure for providing automatic upgrades for distributed systems.

2.1 System Model

We model a distributed system as a collection of objects that communicate via method calls. An object has an identity, a type, and a state. A *type* identifies a behavior for all objects of that type. A *specification* describes that behavior e.g., informally or in some precise mathematical notation. A specification defines an abstract state for objects of the type and defines how each method of the type interacts with that state (in terms of method preconditions and postconditions). An object is an instance of a *class* that defines how the object implements its type.

Because nodes and the network may fail, objects are prepared for any remote method call to fail. Systems based on remote procedure calls [99] or remote method invocations [79] map easily to this model. Extending the model to general message-passing systems is future work.

A portion of an object's state may be persistent, e.g., it may reside on disk or on other nodes. Objects are prepared for failure of their node, and such failure may occur at any point in an object's computation (i.e., the object may not be shut down cleanly). When the node recovers, the object reinitializes itself from the persistent portion of its state.

The model allows for multiple objects per node, but to simplify our discussion, we assume just one object per node. Thus, each node runs a top-level class—the class that implements its object. When there are multiple objects per node, each object's class may be upgraded independently.

We assume class definitions are stored in well-known repositories and define the full implementation of an object, including its subcomponents and libraries. Modern software packaging schemes like RPM [25] and APT [1] satisfy this assumption. Different nodes are likely to run different classes, e.g., clients run one class, while servers run another.

2.2 Upgrade Model

The *schema* of a system is a type-correct set of classes for the nodes in the system, i.e., each class in a schema relies only on the types of other classes in that schema. An upgrade defines a new schema and a mapping from the old (preceding) schema's classes to the new schema's classes. Some old classes are simply carried forward into the new schema, but other old classes are replaced by new classes; each such replacement is a *class upgrade*.

We associate a *version number* with each schema. The initial schema has version number one (1). Each subsequent schema has the succeeding version number. Thus, an upgrade moves the system from one version to the next.

A class upgrade defines how to replace instances of an old class with instances of a new class. A class upgrade has six components, identified as ⟨*oldClassID, newClassID, TF, SF, pastSO, futureSO*⟩. *OldClassID* identifies the class that is now obsolete; *newClassID* identifies the class that is to replace it. *TF* identifies a *transform function* that generates an initial persistent state for the new object from the persistent state of the old object. *SF* identifies a *scheduling function* that tells a node when it should upgrade. *PastSO* and *futureSO* identify classes for *simulation objects* that enable nodes to interoperate across versions. A *futureSO* object allows a node to support the new class's behavior before it upgrades. A *pastSO* object allows a node to support the old class's behavior after it upgrades.

This design allows upgraded nodes to interoperate with non-upgraded nodes. In fact, a series of simulation objects can enable nodes separated by several versions to interoperate, which is important when upgrades happen slowly or when nodes may be disconnected for long periods. Our design is modular: defining the components of a class upgrade requires an understanding of just the old and new classes, regardless of how many legacy versions exist and how many versions separate communicating nodes.

While each class upgrade has six components, many of these can be omitted for most upgrades. A transform function is only needed when an upgrade changes how an object organizes its persistent state. Simulation objects are only needed when an upgrade changes an object's type, so depending on the kind of change, the upgrade can omit the past SO or future SO or both. Scheduling functions cannot be omitted, but they are simple to implement, and it is often possible to select a "stock" SF from a library.

Our discussion assumes that a class upgrade causes all nodes running the old class to switch to the new class. We could, however, provide a filter that restricts a class upgrade to only some nodes belonging to the old class. Filters are useful to upgrade nodes selectively, e.g., to optimize those nodes for their environment or hardware. Providing filters is non-trivial: they must have enough expressive power to be useful, but processing them must not delay upgrades. We do not explore these issues in this thesis; this is an area of future work.

Class upgrades enable a system to replace existing classes with new ones, and with filters, this is enough to restructure a system in arbitrary ways. One can also introduce a new class (that's not a replacement for an existing class) by initializing a node with that class directly.

2.3 How an Upgrade Happens

This section introduces our infrastructure for providing automatic upgrades and describes how an upgrade happens.

The upgrade infrastructure is invisible to the system being upgraded. The infrastructure disseminates information about upgrades to nodes in the system, causes nodes to upgrade their software at appropriate times, and enables nodes running different versions to interoperate. The infrastructure consists of four kinds of components, as illustrated in Figure 2-1: an *upgrade server*, per-node *upgrade layers*, a *software distribution network*, and an *upgrade database*.

The *upgrade server* stores a *configuration* that identifies the minimum active version, the the initial schema (the classes for version 1), and the components of all the class upgrades:

$$\begin{aligned}
\mathit{configuration} &= \langle \mathit{minVersion}, \mathit{initialSchema}, \mathit{upgrade}{*} \rangle \\
\mathit{initialSchema} &= \mathit{classID}+ \\
\mathit{upgrade} &= \langle \mathit{version}, \mathit{classUpgrade}{+} \rangle \\
\mathit{classUpgrade} &= \langle \mathit{oldClassID}, \mathit{newClassID}, \mathit{TF}, \mathit{SF}, \mathit{pastSO}, \mathit{futureSO} \rangle
\end{aligned}$$

This is a simplified description of a configuration; we present the full details in Appendix A.

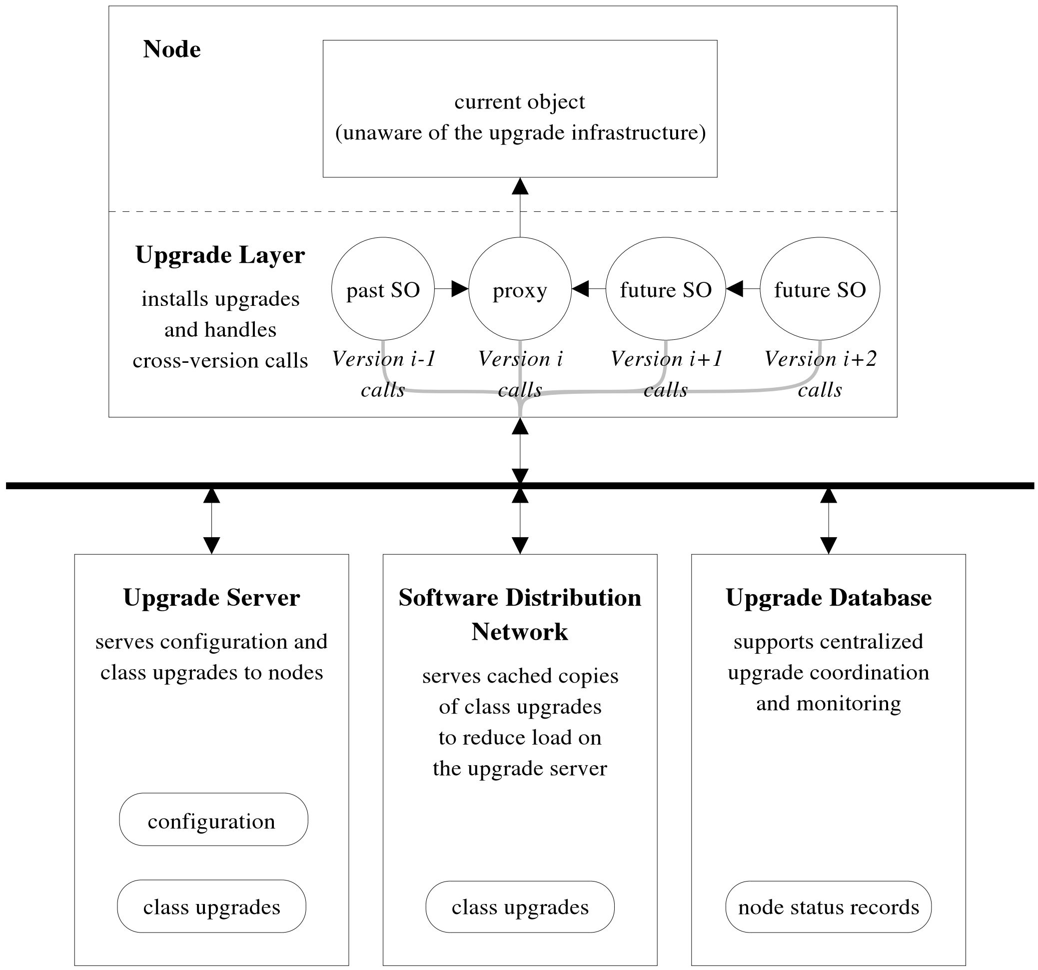


Figure 2-1: *The upgrade infrastructure. The components communicate over a network (bold line); arrows indicate the direction of (possibly remote) method calls. The node is running an instance of its class for version i and SOs for versions i-1, i+1, and i+2.*

The configuration is small: it simply identifies the components of class upgrades—it does not contain any code. The configuration can only be defined by a trusted party, called the *upgrader*, who must have the appropriate credentials to modify the configuration. Nodes are required to support calls for any version from *minVersion* up to the version of the latest upgrade: these are the *active* versions of the system. The upgrader defines a new version by adding an *upgrade* to the configuration. The upgrader retires old versions by increasing *minVersion*. Given a configuration, it is easy to determine the schema for any version: start with the initial schema and, for each successive version, replace classes as defined by the class upgrades for that version.

Our model allows multiple systems to coexist on the same set of nodes. Each system has its own configuration, and these configurations may be stored on the same or on different upgrade servers.

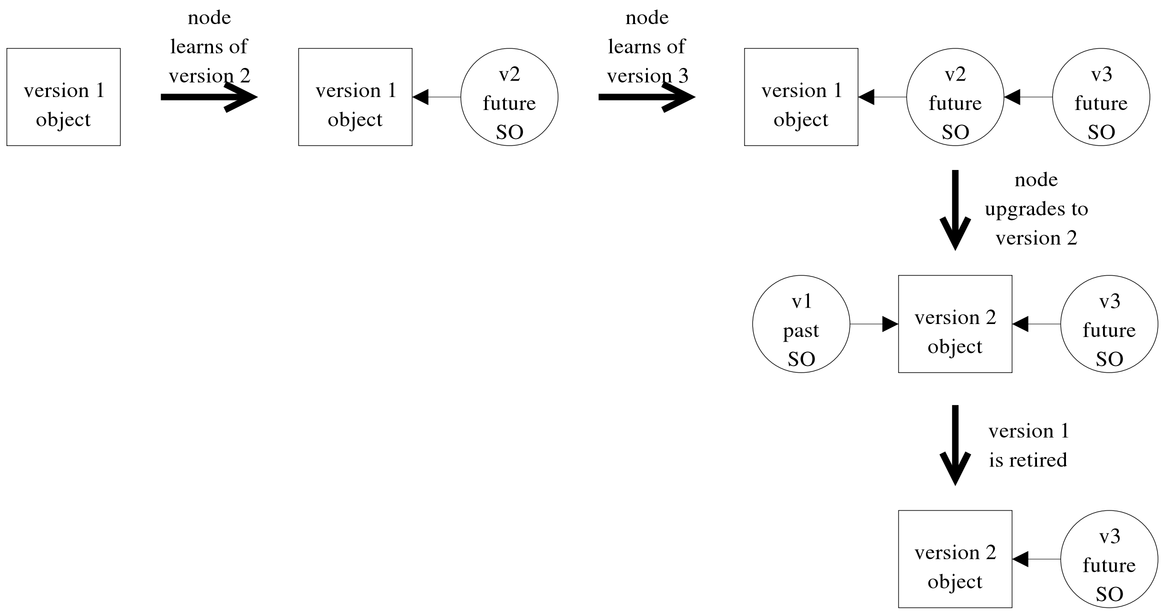


Figure 2-2: *How an upgrade happens, presented as a sequence of states of a node. Large arrows indicate state transitions. In each state, the box is the current object of the node, and the circles are SOs. Newer versions are to the right. Objects may delegate calls as indicated by the small arrows. The node handles calls for versions 1, 2, and 3; but only the "node upgrades to version 2" transition actually disrupts node service.*

Each node in a system is running a *current version*, which is the version of the last upgrade installed on that node (or the initial version, if the node has not installed any upgrades). The node's *current object* is an instance of its *current class*; this is the new class of the last upgrade that the node installed (or its initial class).

A node's *upgrade layer* labels outgoing calls made by an object with the object's version number: calls made by the current object are labeled with the node's current version number, and calls made by an SO are labeled with the SO's version number. The upgrade layer dispatches incoming calls as follows. Calls made by clients running the node's current version are handled by the node's current object.[1] Calls made by clients running newer versions than the node's current version are handled by future simulation objects. Calls made by clients running older versions than the node's current version are handled by past simulation objects.

We now discuss how nodes discover and install new upgrades. Figure 2-2 depicts this process.

The upgrade layer discovers new versions by periodically downloading the configuration (a small file) from the upgrade server and checking whether it defines an upgrade for the next version after the node's current version. If so, the upgrade layer checks whether the upgrade for the new

[1]Calls at the node's current version may actually be handled by another object that implements the node's current type. We will discuss the details of dispatching in Chapter 4.

version includes a class upgrade whose old class matches the node's current class. If so, the node is affected by the upgrade. Otherwise, the node is unaffected and immediately advances its version number. We'll explain how a node that's several versions behind "catches up" in Section 2.3.1.

The upgrade layer also discovers new versions by gossip: it examines the version numbers of incoming calls and periodically exchanges the newest version number it has encountered with other nodes. When the upgrade layer encounters a new version number, it downloads the latest configuration and checks it as described above.

If a node is affected by an upgrade, its upgrade layer fetches the class upgrade components and new class implementation from the *software distribution network*. The network enables the system to disseminate upgrades rapidly to all the nodes in the system without overloading the upgrade server. Reducing load is important, because each upgrade may include several large files, and there may be thousands of nodes attempting to download these files simultaneously.

Once the upgrade layer has downloaded the components of a class upgrade, it verifies the upgrade's authenticity (by checking digital signatures on the components) and then installs the class upgrade's future SO, which lets the node support (some) calls at the new version. The upgrade layer dispatches incoming calls labeled with the new version to the future SO. This SO can delegate to (i.e., call methods of) the object for the previous version, which may be another SO or may be the current object. The node may install additional future SOs for later upgrades, so there could be a chain of SOs, each member of which implements the type of its upgrade's new class. The chain ends at the node's current object.

After installing the SO, the upgrade layer invokes the class upgrade's scheduling function, which runs in parallel with the node's current object, determines when the node should upgrade, and signals the upgrade layer at that time. The scheduling function may access a centralized *upgrade database* to coordinate the upgrade schedule with other nodes and to enable human operators to monitor and control upgrade progress.

In response to the scheduling signal, the upgrade layer shuts down the node (the current object and all the SOs). The upgrade layer then installs the new class implementation and runs the transform function to convert the node's persistent state to the representation required by new class. The upgrade layer then discards the future SO and installs the past SO, which implements the old type. The upgrade layer then causes the node to start running an object of the new class, which recovers from the newly-transformed persistent state. Finally, the upgrade layer notifies the upgrade database that its node is running the new version.

Like future SOs, past SOs can implement their calls by delegating, but they delegate to the object of the next newer version rather than the previous one. After several upgrades, there could be a chain of past SOs, and again the chain ends at the node's current object.

Once all of the nodes in the system have upgraded, the upgrader can retire the old version by increasing the minimum active version number in the configuration. When nodes learn of this update, they discard their past SOs for any versions less than the minimum active version. This can be done lazily, since keeping past SOs around does not affect the behavior or performance of later versions. If a version is retired before all the nodes have upgraded beyond that version, nodes running the retired or earlier versions will be unable to communicate with nodes running later versions. This problem is easy to avoid, since the upgrader can query the upgrade database to determine whether any nodes are still running old versions.

2.3.1 Catching Up

If a node is several versions behind the latest version, it may have several upgrades pending. We want the node to start supporting these newer versions as soon as possible, because other nodes may have upgraded and, if so, will rely on the newer versions. Therefore, we allow a node to download and install the future SOs for all its pending upgrades before it starts the scheduling function for its next upgrade.

To determine which upgrades are pending, the node must know which class it will be running at each future version. The algorithm is as follows: the node starts with its current class c and current version v. The node checks the configuration for an upgrade in version $v + 1$ that replaces class c with some new class c'. If such an upgrade exists, the node downloads and installs the future SO for that upgrade, which allows the node to simulate the type of c'. The node repeats this process, now assuming that its current version is $v + 1$ and its current class is c' (if there was an upgrade) or c (otherwise). The node continues this process until there are no newer versions defined in the configuration.

2.4 Upgrade Components

There are three kinds of upgrade components (beside class definitions): simulation objects, transform functions, and scheduling functions. This section introduces each in turn; Chapters 4, 5, and 6 discuss each in detail.

2.4.1 Simulation Objects

Simulation objects (SOs) are adapters defined by the upgrader to enable communication between nodes running different versions. Simulation is necessary when nodes upgrade asynchronously, since nodes running older versions may make calls on nodes running newer versions, and vice versa. It is important to enable simulation in both these directions, because otherwise a slow upgrade can partition upgraded nodes from non-upgraded ones (since calls between those nodes will fail). Simulation also simplifies software development by allowing implementors to write their software as if every node in the system were running classes in the same schema.

SOs are wrappers: they delegate (most of) their behavior to other objects. This means that SOs are simpler to implement than full class implementations, but they are also slower than full implementations and may not be able to implement the full type (i.e., SOs may have to reject calls that they cannot implement correctly—we discuss when this happens in Chapters 3 and 4). If a new version does not admit good simulation, the upgrader may cause the upgrade to happen as quickly as possible (and cause SOs to reject all calls), at the expense of disrupting service while the upgrade happens.

At a given time, a node may contain a chain of past SOs and a chain of future SOs, as depicted in Figure 2-1. An SO may call methods only on the next object in the chain; it is unaware of whether the next object is the current object or another SO. An SO can have its own state; this means SOs are more powerful than "translators" used in previous approaches [48, 81, 93, 97, 102].

When a node receives a call, its upgrade layer dispatches the call to the object that implements the version indicated in that call (or rejects the call if it is for an inactive version). The infrastructure ensures that an object exists for every active version by dynamically installing future SOs for new versions and by keeping past SOs until their versions are retired.

Chapter 3 presents a model for nodes that support multiple types simultaneously and explains how to reason about computations on such nodes. Chapter 4 discusses various techniques for realizing this model using simulation objects.

2.4.2 Transform Functions

Transform functions (TFs) are procedures defined by the upgrader to convert a node's persistent state from the representation required by its current class to the representation required by a new class. We allow a node to simulate the new type before the TF runs, so the TF must take into account

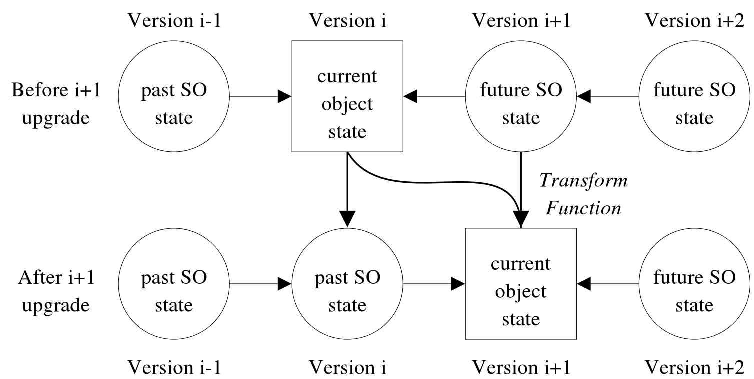


Figure 2-3: *State transform for a node upgrade from version i to i+1*

the persistent state of the future SO, as illustrated in Figure 2-3. The job of the TF is to produce a state for the new object whose value reflects the abstract state of the future SO at the moment the TF runs. The TF must also produce a state for the past SO whose value reflects the abstract state of the old object at the moment the TF runs.

Chapter 5 discusses transform functions and recovery, including recovery from failures that occur while a TF is running.

2.4.3 Scheduling Functions

Scheduling functions (SFs) are procedures defined by the upgrader to tell nodes when to upgrade. SFs run on the nodes themselves, which lets them respond quickly to changing conditions—e.g., to avoid upgrading a replica if another one fails—and decide when to upgrade even if the network is slow or unavailable.

Scheduling functions control the order and rate of node upgrades, so they can affect a system's availability, fault-tolerance, and performance during an upgrade. For example, a schedule that upgrades nodes as quickly as possible may cause severe service disruption, but this may be appropriate if the upgrade closes a dangerous security hole or occurs during off-peak hours. On the other hand, a gradual schedule can minimize service disruption, but this increases the period of time during which nodes running different version must interoperate.

Many scheduling functions require information about the nodes on which they run or about the system as a whole. For example, an SF may need to know the current time and load of a node to avoid disrupting client activity. Or the SF might check with a central database to determine whether its node is allowed to upgrade. Our goal is to give the upgrader as much flexibility as possible in defining upgrade schedules.

We discuss scheduling functions in more detail in Chapter 6, including guidelines for designing good scheduling functions and several examples.

2.5 Example

We present an example upgrade to demonstrate the features of our model. The example system is a distributed file system, like SFS [76] or AFS [61]. The system has two classes of nodes: clients and servers. Servers store and control access to files. Clients use files by interacting with servers.

In version 1 of our system, servers control access to files using Unix-style permissions. That is, a server keeps a set of nine bits, Or, Ow, Ox, Gr, Gw, Gx, Wr, Ww, Wx, for each file that indicates whether the file owner (O), file group (G), and the rest of the world (W) can read (r), write (w), or execute (x) the file. Permissions are part of a server's persistent state; they are stored in metadata blocks in the file system.

Unix-style permissions are adequate for small collaborative settings, but they are not expressive enough for larger systems. For example, permissions cannot designate a file as writable for one group of users, read-only for another group, and inaccessible to the rest.

To provide greater control over permissions, file systems like AFS [61] and SFSACL [64] keep an access control list for each file. That is, each file has a mapping from user or group names to a set of permissions.

In version 2 of our system, servers keep an access control list for each file. Access control lists are part of a server's persistent state; each ACL is stored in the first 512 bytes of the file it protects. This change does not affect how clients use files, because the server hides ACLs from clients, i.e., when a client reads the first 512 bytes of a file, it sees the file's data, not its ACL. However, this change does affect how clients manage file permissions, so the protocol between clients and servers must change.

Thus, the upgrade from version 1 to version 2 consists of two class upgrades: the first changes the file server implementation from PermServer (a server that uses permissions) to AclServer (a server that uses access control lists), and the second changes the client implementation to support management of access control lists.

2.5.1 Model

We do not model the clients, because they have no methods.

PermServer has the following methods (the client's identity is an implicit parameter to all server methods):

getPerms(f) Returns the owner, group, and permission bits for file f.

setPerms(f, owner, group, bits) Sets the owner, group, and permission bits for file f, or throws NoAccess if the client is not f's (current) owner.

canAccess(f, mode) Returns true iff the client can access file f in mode mode according to f's permissions.

We omit the methods for creating, reading, writing, and removing files from this model because they are unaffected by our example upgrade (these methods use canAccess to check whether the client is allowed to do the requested operation).

AclServer has the following methods:

getACL(f) Returns the access control list for file f.

setACL(f, acl) Sets the access control list for file f, or throws NoAccess if f's ACL does not grant the client permission to modify the ACL.

canAccess(f, mode) Returns true iff the client can access file f in mode mode according to f's access control list.

The rest of the server's methods (those for manipulating files) are unchanged.

Now we consider the components of the two class upgrades.

2.5.2 Scheduling Functions

We have to define scheduling functions for both the client and server class upgrades.

We want to avoid disrupting service, so the client scheduling function should attempt to avoid signaling the upgrade layer while the user is active. The SF can do this by signaling late at night or while the client has had little activity for a long time. Alternately, the SF could periodically display a dialog box to the user requesting that the client be allowed to upgrade.

The server scheduling function also needs to avoid disrupting service, but if there is just one server, the best the SF can do is schedule the server upgrade when there is low client activity.

However, if servers are replicated, we can use a *rolling upgrade* [16, 33, 102] that upgrades just one replica at a time. In this case, the SF signals the upgrade layer when all other replicas with lower IP addresses have upgraded. Since IP addresses define a total order on servers, this SF implements a rolling upgrade.

2.5.3 Simulation Objects

The server upgrade changes the server's type. Our scheduling functions allow clients to upgrade before servers or vice versa, so we must use simulation objects to enable upgraded clients to use non-upgraded servers and non-upgraded clients to use upgraded servers. This means we need two SOs: a future SO that implements AclServer while the node is running the PermServer implementation and a past SO that implements PermServer while the node is running the AclServer implementation.

The challenge in defining these SOs is that some ACLs cannot be expressed as permissions. We explain how to reason about this challenge and define the SOs in Section 3.5.

2.5.4 Transform Functions

The client class upgrade does not change the client's persistent state, so it requires no transform function. However, the server class upgrade requires a TF to convert the permissions for each file to access control lists. Permissions are stored in metadata blocks in the file system, whereas ACLs are stored in the first 512 bytes of the files themselves [64].

Therefore, the TF needs to change the representation of every file on the server. For each file in the file system, the TF copies the file's data to scratch space, reads the file's permissions from the metadata blocks, converts these to an ACL (as described in Section 3.5), writes the ACL to the beginning of the file, then writes the file's data after the ACL. We have implemented this TF, as described in Section 8.2.2.

This TF is slow, because it needs to read and write each file's data twice (to the scratch space and back to the file). This takes a long time for large file systems, and the server is unavailable while the TF is running. One way to reduce this downtime is to allow the server to recover immediately and run the TF incrementally, e.g., transform each file only as it is accessed. Supporting incremental transforms is future work; we discuss how to support them in Section 10.3.1.

Chapter 3

Specifying Upgrades

Our approach allows nodes to upgrade asynchronously and yet communicate, even though upgrades may make incompatible changes. We accomplish this by augmenting nodes to support multiple types simultaneously. This enables a node's non-upgraded clients to use the node's pre-upgrade type and enables upgraded clients to use the node's post-upgrade type.

This chapter explains how to specify the relationship between a node's pre-upgrade and post-upgrade types as part of defining an upgrade. This specification guides the design of the simulation objects and transform function for an upgrade. The specification also tells clients what changes to expect when they upgrade from one version to the next and switch from using the pre-upgrade to the post-upgrade type.

The definitions in this chapter are informal, but we aim for them to be precise enough to enable developers to reason about programs and upgrades. Formalizing our model is future work.

3.1 Specifications

There are various ways of defining specifications; in this thesis, we use informal specifications in the *requires / effects* style [72] to define the preconditions and postconditions of a method.

- The *requires* clause defines constraints on the arguments to a method and the state of the object before the method runs.

- The *effects* clause describes the behavior of the method for all inputs not ruled out by the *requires* clause: the outputs it produces, the exceptions it throws, and the modifications it makes to its inputs and to the state of the object.

class IntSet	*An IntSet is a mutable, unbounded set of integers*
IntSet()	*effects:* $this = \{\}$
void insert(x)	*effects:* $this_{post} = this_{pre} \cup \{x\}$
void delete(x)	*effects:* $x \in this_{pre} \Rightarrow this_{post} = this_{pre} - \{x\}$, *else throws NoSuchElementException*
boolean contains(x)	*effects: returns* $x \in this$

Figure 3-1: *Specification for* IntSet

Figure 3-1 gives a specification for type IntSet (a set of integers) in this style. We use $this_{pre}$ to denote the pre state of the object and $this_{post}$ to denote the post state. All examples of method specifications in this thesis are total, so none has a *requires* clause.

3.1.1 Failures

Each method call terminates either normally or by throwing an exception. Furthermore, we assume that any call can terminate by throwing a special exception that indicates *failure*. Such termination happens, e.g., if a remote method call fails because the target node is unreachable. Failures are unpredictable can happen at any time. This model is common in distributed object systems: Java RMI [79] allows any remote method invocation to fail by throwing `RemoteException`, and Sun RPC [99] allows any remote procedure call to fail with error codes indicating the unavailability of a server or individual procedures.

We extend the notion of failure to include cases when a node is unable to handle a call correctly because the version of the caller is different from that of the receiver. These failures appear to the caller as node failures, so the caller may try an alternate node or a workaround, or it may wait and try the call again later. Of course, such failures and delays disrupt service, so we want to avoid causing calls to fail when possible.

3.1.2 Subtypes

We review behavioral subtyping [73] briefly here.

One type is a *subtype* of another if objects of the subtype can be substituted for objects of the supertype without affecting the behavior of callers that expect the supertype. Subtypes that satisfy this *substitution principle* support three properties (as stated in [72]):

Signature Rule The subtype objects must have all the methods of the supertype, and the signatures of the subtype methods must be *compatible* (contravariant argument types, covariant return and exception types) with the signatures of the corresponding supertype methods.

Methods Rule Calls of subtype methods must "behave like" calls to the corresponding supertype methods, i.e., subtype methods may weaken the precondition and strengthen the postcondition of the corresponding supertype methods.

Properties Rule The subtype must preserve invariants and history properties that can be proved about supertype objects.

Invariants are "properties true of all states" of an object, and history properties are "properties true of all sequences of states" of an object [73]. History properties define how an object evolves over time, and they are derived from the specification of the object's type. For example, if we remove the delete method from *IntSet*, then we can derive the property that later sets are always supersets of earlier ones (regardless of the intervening method calls). Clients can rely on this property when reasoning about programs that use this type, e.g., they know that once an element is in a set, it will always be in the set.

3.2 Class Upgrades

A class upgrade concerns two classes, an *old class* and a *new class*, and causes each instance of the old class to be replaced by an instance of the new class. The old and new classes each implement their respective types.

We can classify class upgrades based on how the old and new types are related. There are four possibilities:

Same type The old and new types are the same.

Subtype The new type is a subtype of the old type.

Supertype The new type is a supertype of the old type.

Unrelated The old and new types are incomparable.

We say an upgrade is *compatible* if the new type is the same type or a subtype of the old type; otherwise it is *incompatible*.

Same type upgrades are likely to be common; they correspond to patches that change internal algorithms of nodes without affecting their types. Subtype upgrades are also likely to be common; they correspond to minor releases that introduce new features. Unrelated upgrades are likely to be the next most common; they correspond to major, incompatible software changes. Supertype upgrades are probably rare, as they remove behaviors without providing replacements.

3.3 Requirements

At a given time, a node may support multiple types. The node implements its *current type* using its current object. The node may also simulate several old types (of classes that it upgraded from in the past) and several new types (of classes that it will upgrade to in the future). But clients do not know whether they are using the current type or a simulated one; they simply believe they are using an object of the type they expect.

Our goal is for upgrades to be *transparent* to clients of all versions [78, 94], i.e., clients should not notice when a node upgrades and changes its current type. Furthermore, we want to enable clients to reason about their programs, not only when they are making calls to nodes that are running their own version, but also when they are making calls to nodes that are running newer or older versions than their own, when they are interacting with other clients that are using the same node via a different version, and when the client itself upgrades and resumes using a node it was using before it upgraded. Essentially, we want nodes to provide service that makes sense to clients, and we want this service to make sense across upgrades of nodes and clients.

This section defines requirements for how upgrades must be defined so that clients can reason about the behavior of nodes that support multiple types. These requirements provide an *intuition* for what an upgrade definition must provide; we will explain in Section 3.4 precisely what additional information is needed.

We think of a node that implements multiple types as having an object for each type. (The node may not actually implement each type with a separate object; how a node actually implements its types is the subject of Chapter 4.) We can refer to these objects by version, e.g., O_2 is the object that implements the node's type for version 2. When we are discussing a specific class upgrade, O_{old} is the object that implements the *old type* (T_{old}, the type of the class replaced by that upgrade), and O_{new} is the object that implements the *new type* (T_{new}, the type of the replacement class). We sometimes refer to O_{old} and O_{new} as the *old object* and the *new object*, respectively.

Clearly, we require the following:

Specification Requirement The object for each version must implement its type.

This ensures that a client's call behaves as expected by that client.

However, we also need to define the effects of *interleaving*. Interleaving occurs when different clients running different versions interact with the same node. For example, a call to a node's current type (made by a client running the node's current version) may be followed by a call to a past type (made by a client running a past version), which may be followed by a call to a future type, and so on.

To be more precise about what we mean by interleaving, we introduce the notion of the *computation* at a node. A computation is a series of *events*; each event is either the execution of a method on some type implemented by the node, the addition of a new type, the removal of an old type, or a node upgrade (which changes the current type). For now we assume the events in a computation occur one-at-a-time in some serial order; we discuss how this happens in Section 4.6.

By interleaving, we mean computations like:

O_1*.m(args)*; O_1*.m(args)*; [version 2 introduced];
O_1*.m(args)*; O_2*.p(args)*; [node upgrades from 1 to 2];
O_1*.m(args)*; O_2*.p(args)*; [version 1 retired];
O_2*.p(args)*; O_2*.p(args)*;

where between the introduction of version 2 and the node upgrade and between the node upgrade and the retirement of version 1 there can be an arbitrary sequence of calls to O_1 and O_2. A node may support more than two types simultaneously, in which case calls to all of the supported types can be interleaved.

The problem we face is how to make sense of what is happening in such computations. The objects O_1 and O_2 (and so on) are not independent: they share a single identity, so calls made to one must reflect the effects of calls made to the others. For example, when a node implements two versions of a file system protocol simultaneously, modifications to a file made by one client (using one protocol version) must be visible to other clients (using the other protocol version).

In general, an upgrade must *specify* the effect on the old object (O_{old}) of calls to the new object (O_{new}), and vice versa. For example, data written to a file via O_{new} must be visible when the file is later read via O_{old}; the specification of the upgrade from T_{old} to T_{new} must specify exactly what the effect of a modification to O_{new} is on O_{old}, and vice versa. We explain how to specify these effects in Section 3.4.

We require the following:

Sequence Requirement Each event in the computation at a node must reflect the effects of all earlier events in the computation at that node in the order they occurred.

In the simple case of an object with just two types, this requirement means method calls to one type must reflect the effects of calls made via the other, and vice versa. If the method is an observer, its return value must reflect all earlier modifications made via either type. If the method is a mutator, its effects must be visible to later observations made via either type. When the node upgrades and its current type changes, observations made via either type after the upgrade must reflect the effects of all modifications made via either type before the upgrade. We explain how upgrade definitions guarantee the sequence requirement in general in Section 3.6.

The two requirements stated above—the specification requirement and the sequence requirement—can be overconstraining: it may not be possible to satisfy them both for all possible computations (we'll explain why in Sections 3.4.5 and 3.6). When this happens, we resolve the problem by *disallowing* calls. The system causes any disallowed call to fail (i.e., to throw a failure exception). We meet the requirements above essentially by ruling out calls that would otherwise cause problems.

Disallowing takes advantage of the fact that any call can fail, so clients won't be surprised by this. We can disallow whole methods, in which case any call to those methods fail, or we can disallow at a finer granularity, e.g., based on the arguments of a call.

We require that calls to the current type are never disallowed:

Disallow Constraint Calls to the current type must not be disallowed.

The rationale for this constraint is that the current type provides the "real behavior" of the node, so it should not be affected by the node's support for other versions. Given this constraint, we want to disallow as few calls as possible so as to provide the best possible service to all clients.

3.4 Defining Upgrades

This section explains what is needed to define an upgrade.

An upgrade deals with two objects—the new one (O_{new}) and the old one (O_{old}). The upgrade definition will include an *invariant*, $I(O_{old}, O_{new})$, that relates the old and new objects throughout the computation, i.e., when O_{new} is introduced, after each method call to O_{old} or O_{new}, and until O_{old} is retired. For all but same-type upgrades, the upgrade definition will also include a *mapping function (MF)* that defines an initial state for O_{new} given the state of O_{old} when T_{new} is introduced. Finally, the definitions of unrelated-type upgrades must also *explicitly* specify the effects of calls to O_{new} on O_{old} (and vice versa); these specifications are given in the form of *shadow methods*.

The following sections explain how to define each of the four kinds of upgrades: same type, subtype, supertype, and unrelated.

3.4.1 Same Type

If the new type is the same as the old one, no additional information is required for the upgrade: O_{old} and O_{new} behave like a single object. We want the following invariant to hold throughout the computation:

$$O_{old} = O_{new} \tag{3.1}$$

where in this context O_{old} and O_{new} refer to the abstract states of those objects. To satisfy this invariant, the effect of a method call on one of the objects (e.g., O_{new}) must be reflected on the other (e.g., O_{old}) just as defined by the specification of that method.

For example, consider an upgrade that replaces an old implementation of IntSet (Figure 3-1, page 33) with a new one (e.g., because the new implementation is more efficient). Our invariant means that if a client calls O_{old}.insert(x), a subsequent call O_{new}.contains(x) will return true (provided neither call fails and no intervening calls remove x).

3.4.2 Subtype

If the new type is a subtype of the old one, we want the following invariant to hold:

$$O_{old} = AF(O_{new}) \tag{3.2}$$

where AF is the abstraction function that maps the abstract state of the subtype to that of the supertype [73]. The invariant must be defined this way because of the subtype relationship: when clients of O_{old} upgrade and start using O_{new}, they expect to see values that are consistent with what they saw before they upgraded.

All that is needed in the upgrade (besides the specifications of the two types) is one additional piece of information, the *mapping function (MF)*:

$$MF : O_{old} \rightarrow O_{new} \tag{3.3}$$

Given an object of the old type, MF defines a related object of the new type. The mapping function is used to define the state of the new object when it is first introduced. It must establish our invariant, i.e., it must respect the abstraction function:

$$O_{old} \quad = \quad AF(MF(O_{old})) \tag{3.4}$$

Method calls to either object must preserve the invariant. This means the effect of a call made via the old object (the supertype) is reflected on the new object according to the subtype's specification of that method. By the definition of a subtype, this preserves our invariant. The effect of a call made via the new object (the subtype) is reflected on the old object by applying the method to the new object then applying the abstraction function. The result is the new value for the old object, and this clearly satisfies our invariant.

Since the new object may be introduced at an arbitrary time, we would like to get the same result regardless of when we run the mapping function. This is not automatic—we give an example below of how it might not be the case—so the mapping function must be defined to satisfy this property:

$$MF(O_{old}.m(args)) \quad = \quad MF(O_{old}).m(args) \tag{3.5}$$

Here, m is a method of T_{old}, and $O_{old}.m(args)$ is the state of O_{old} after running *m(args)* (not the return value of *m(args)*). This property ensures clients cannot tell when a node introduces the new object. While this is not strictly necessary, we believe it is always possible to satisfy this property for subtype upgrades. Proving this conjecture is future work.

Example: Replace IntSet with ColorSet

Consider an upgrade that replaces IntSet (Figure 3-1, page 33) with ColorSet (Figure 3-2, page 41). This example is analogous to an upgrade that adds a new property to files in a file system or adds a new column to a table in a database.

ColorSet (O_{new}) is a subtype of IntSet (O_{old}) under this abstraction function:

$$O_{old} \;=\; AF(O_{new}) \;\;=\;\; \{\, x \mid \langle x, c \rangle \;\in\; O_{new} \,\} \tag{3.6}$$

The mapping function works in the opposite direction. It specifies an initial ColorSet given an IntSet:

$$O_{new} \;=\; MF(O_{old}) \;\;=\;\; \{\, \langle x, \text{blue} \rangle \mid x \;\in\; O_{old} \,\} \tag{3.7}$$

This MF specifies that the initial ColorSet has the same set of integers as the IntSet, and the initial color for each element is blue. This choice of color is arbitrary: we could have assigned a different color to each integer, or we could have assigned random colors, or we could have used a special value that indicates that the color is undefined. Any of these satisfies our invariant, because applying the abstraction function to any ColorSet they produce yields the original IntSet. But using blue means the MF satisfies property (3.5) (because ColorSet.insert adds new elements with the color blue), so clients cannot tell which elements were inserted after the MF ran. Any other MF definition would not satisfy (3.5).

After running an IntSet method, the ColorSet post-state is the result of running ColorSet's version of that method. For example, if a client calls O_{old}.insert(x), a subsequent call O_{new}.getColor(x) will return blue (assuming neither call fails and no intervening call changes the color of x).

After running a ColorSet method, the IntSet post-state is the result of applying the abstraction function to the ColorSet post-state. For example, if a client calls O_{new}.insertColor(x, green), a subsequent call O_{old}.contains(x) will return true (assuming neither call fails and no intervening call removes x).

3.4.3 Supertype

The new type is a supertype of the old one, so we want the reverse of the previous invariant:

$$AF(O_{old}) \;\;=\;\; O_{new} \tag{3.8}$$

As in the subtype case, the upgrade must provide the specifications for the two types and a mapping function. The mapping function maps the old object to the new and must satisfy our invariant, so we must use $MF = AF$, i.e., the mapping function is just the abstraction function.

class ColorSet	*A* ColorSet *is a mutable, unbounded set of colored integers; integers are unique:* $\langle x, c\rangle \in this \ \wedge \langle x, c'\rangle \in this \ \Rightarrow \ c = c'$
ColorSet()	*effects:* $this = \{\}$
void insert(x)	*effects:* $\neg\exists \langle x, c\rangle \in this_{pre} \Rightarrow this_{post} = this_{pre} \cup \{\langle x, \text{blue}\rangle\}$
void delete(x)	*effects:* $\exists \langle x, c\rangle \in this_{pre} \Rightarrow this_{post} = this_{pre} - \{\langle x, c\rangle\}$, *else throws NoSuchElementException*
boolean contains(x)	*effects: returns* $\exists \langle x, c\rangle \in this$
void insertColor(x, c)	*effects:* $\neg\exists \langle x, c'\rangle \in this_{pre} \Rightarrow this_{post} = this_{pre} \cup \{\langle x, c\rangle\}$
color getColor(x)	*effects:* $\exists \langle x, c\rangle \in this_{pre} \Rightarrow$ *returns c, else throws NoSuchElementException*

Figure 3-2: *Specification for* ColorSet

The effects of methods are symmetric with the subtype case. The effect of a call on the old object (the subtype) is reflected on the new object by applying the method to the old object then applying the abstraction function. The effect of a call made via the new object (the supertype) is reflected on the old object according to the subtype's specification of that method. These definitions mean the MF automatically satisfies property (3.5).

As an example, consider an upgrade that replaces ColorSet with IntSet. This is symmetric with the example in Section 3.4.2, except in this case, the mapping function is the same as the abstraction function (3.6).

3.4.4 Unrelated Type

In this case, there is no subtype relationship between the two types. Instead, the upgrader needs to define their relationship.

The first step in defining the relationship between T_{old} and T_{new} is to define the invariant, $I(O_{old}, O_{new})$, that relates the old and new objects throughout the computation, i.e., assuming $I(O_{old}, O_{new})$ holds when a method on one of the objects starts, $I(O_{old}, O_{new})$ also holds when the method returns. The invariant is likely to be obvious to the upgrader. For example, if O_{old} and O_{new} are file systems, an obvious invariant is that the new and old file systems contain the same files (although some file properties may differ).

The invariant must be *total*, i.e., for each legal state O_{new} of T_{new}, there exists some legal state O_{old} of T_{old} such that $I(O_{old}, O_{new})$ holds (and vice versa).

The second step is to define the mapping function; it must establish the invariant, i.e., $I(O_{old}, MF(O_{old}))$. For example, the MF from the old file system to the new one must initialize the new file system with all of the old files, and it must also initialize any new file properties to default values.

I tells us something about what we expect from method calls. In particular, it constrains the behavior of mutators (methods that modify the state of the object). For example, it wouldn't be correct to add a file to O_{new} but not to O_{old}. But I doesn't tell us exactly what effect a mutator on O_{new} should have on O_{old}, or vice versa. This information is given by *shadow methods*.

For each mutator $T_{old}.m$, we specify a related method, $T_{new}.shadowT_{old}\$m$ (reads as "the shadow of T_{old}'s method m"). The specification of $T_{new}.shadowT_{old}\$m$ explains the effect on O_{new} of running $T_{old}.m$. Similarly, for each mutator $T_{new}.p$, we specify a related method, $T_{old}.shadowT_{new}\$p$, that explains the effect on O_{old} of running $T_{new}.p$. (We are assuming here that shadow methods never introduce naming conflicts; clearly other naming conventions could be used instead.)

No shadow methods are required for observers; an observer reflects the abstract state of its object at the moment it runs. And shadow methods are often obvious, e.g., for mutators that both types inherit from a common supertype. Therefore, defining shadow methods need not be very onerous.

We require that a shadow method be able to run whenever the corresponding real method can run. This means the precondition for a shadow method must hold whenever the precondition for the corresponding real method holds:

$$I(O_{old}, O_{new}) \wedge pre_m(O_{old}) \quad \Rightarrow \quad pre_{shadowT_{old}\$m}(O_{new}) \tag{3.9}$$

$$I(O_{old}, O_{new}) \wedge pre_p(O_{new}) \quad \Rightarrow \quad pre_{shadowT_{new}\$p}(O_{old}) \tag{3.10}$$

All examples of method specifications in this thesis are total, so they meet this condition trivially.

Shadow methods must preserve the invariant:

$$I(O_{old}, O_{new}) \quad \Rightarrow \quad I(O_{old}.m(args), O_{new}.shadowT_{old}\$m(args)) \tag{3.11}$$

$$I(O_{old}, O_{new}) \quad \Rightarrow \quad I(O_{old}.shadowT_{new}\$p(args), O_{new}.p(args)) \tag{3.12}$$

This enables us to prove that our invariant holds throughout the computation of a node that implements the old and new types simultaneously. The proof is by induction: the mapping function establishes the base case (when the new type is introduced), and shadow methods give us the inductive step (on each mutation).

Just as in the subtype case, we would like to get the same result regardless of when we run the mapping function. We cannot use property (3.5), because T_{new} may not have all of T_{old}'s methods. Instead, we define this property in terms of the shadows of T_{old}'s methods:

$$MF(O_{old}.m(args)) \quad = \quad MF(O_{old}).shadowT_{old}\$m(args) \tag{3.13}$$

This means clients cannot tell when the new object was introduced. This is not strictly necessary, and in some cases satisfying this property is impractical, e.g., when T_{new}'s history properties are stronger than those of T_{old} (we present an example of this in Section 3.4.5). Therefore, we leave this property as a guideline for designing MFs.

We can model the other three kinds of upgrades—same type, subtype, and supertype—using shadow methods. In a same-type upgrade, the shadow of a mutator is just the mutator itself. In a subtype upgrade, the shadow of a supertype mutator on the subtype is the mutator as specified for the subtype, i.e., $T_{new}.shadowT_{old}\$m = T_{new}.m$. However, the shadow of a subtype mutator on the supertype is not the supertype method; it's the subtype method (as effected on the supertype), i.e., $O_{old}.shadowT_{new}\$p(args) = AF(O_{new}.p(args))$. The supertype upgrade case is symmetric with the subtype upgrade case.

We now consider examples of how to define invariants, mapping functions, and shadow methods for unrelated-type upgrades.

Example: Replace ColorSet with FlavorSet

This upgrade replaces objects of class ColorSet (Figure 3-2, page 41) with objects of class FlavorSet (Figure 3-3, page 44). This example is analogous to an upgrade that changes a property of files in a file system, such as one that changes permission bits to access control lists (Section 2.5).

We begin by choosing an invariant I that we want to hold for each ColorSet (O_{old}) and FlavorSet (O_{new}):

$$\{ x \mid \langle x, c \rangle \in O_{old} \} \quad = \quad \{ x \mid \langle x, f \rangle \in O_{new} \} \tag{3.14}$$

class FlavorSet	*A FlavorSet is a mutable, unbounded set of flavored integers; integers are unique:* $\langle x, f\rangle \in this \wedge \langle x, f'\rangle \in this \Rightarrow f = f'$
FlavorSet()	*effects:* $this = \{\}$
void insert(x)	*effects:* $\neg\exists \langle x, f\rangle \in this_{pre} \Rightarrow this_{post} = this_{pre} \cup \{\langle x, \mathrm{grape}\rangle\}$
void delete(x)	*effects:* $\exists \langle x, f\rangle \in this_{pre} \Rightarrow this_{post} = this_{pre} - \{\langle x, f\rangle\}$, *else throws NoSuchElementException*
boolean contains(x)	*effects: returns* $\exists \langle x, f\rangle \in this$
void insertFlavor(x, f)	*effects:* $\neg\exists \langle x, f'\rangle \in this_{pre} \Rightarrow this_{post} = this_{pre} \cup \{\langle x, f\rangle\}$
flavor getFlavor(x)	*effects:* $\exists \langle x, f\rangle \in this_{pre} \Rightarrow$ *returns f, else throws NoSuchElementException*

Figure 3-3: *Specification for FlavorSet*

This says that the set of integers in O_{old} and O_{new} are the same. Since ColorSet and FlavorSet are both subtypes of IntSet, another way of putting this is that O_{old} and O_{new} always map to the same IntSet.

We could have chosen a stronger invariant, e.g., one that maps colors to flavors:

$$\begin{aligned} \langle x, \mathrm{blue}\rangle \in O_{old} &\Leftrightarrow \langle x, \mathrm{grape}\rangle \in O_{new}, \\ \langle x, \mathrm{red}\rangle \in O_{old} &\Leftrightarrow \langle x, \mathrm{cherry}\rangle \in O_{new}, \\ \langle x, \mathrm{green}\rangle \in O_{old} &\Leftrightarrow \langle x, \mathrm{lime}\rangle \in O_{new}, \\ &\ldots \end{aligned} \tag{3.15}$$

Whereas (3.14) treats colors and flavors as independent properties, (3.15) says these properties are related.

We could also have chosen a weaker invariant than (3.14):

$$\{ x \mid \langle x, c\rangle \in O_{old} \} \subseteq \{ x \mid \langle x, f\rangle \in O_{new} \} \tag{3.16}$$

This invariant allows O_{new} to contain more elements than O_{old}. Weaker invariants give us more flexibility in defining shadow methods, so they typically require fewer disallowed methods than stronger ones (as we'll discuss in Section 3.4.5).

Given the invariant, the next step is to define a mapping function. For invariant (3.14), we might have:

$$O_{new} \;=\; MF(O_{old}) \quad = \quad \{\, \langle x, \text{grape} \rangle \mid x \;\in\; O_{old} \,\} \tag{3.17}$$

As required, this MF establishes I.

Now we can define the shadow methods:

void ColorSet.shadowFlavorSet$insert(x)

effects: $\neg\exists\, \langle x, c \rangle \;\in this_{pre} \;\Rightarrow\; this_{post} \;=\; this_{pre} \cup \{\langle x, \text{blue} \rangle\}$

void ColorSet.shadowFlavorSet$insertFlavor(x, f)

effects: $\neg\exists\, \langle x, c \rangle \;\in this_{pre} \;\Rightarrow\; this_{post} \;=\; this_{pre} \cup \{\langle x, \text{blue} \rangle\}$

void ColorSet.shadowFlavorSet$delete(x)

effects: $\exists\, \langle x, c \rangle \in this_{pre} \;\Rightarrow\; this_{post} \;=\; this_{pre} - \{\langle x, c \rangle\}$,

else throws NoSuchElementException

void FlavorSet.shadowColorSet$insert(x)

effects: $\neg\exists\, \langle x, f \rangle \;\in this_{pre} \;\Rightarrow\; this_{post} \;=\; this_{pre} \cup \{\langle x, \text{grape} \rangle\}$

void FlavorSet.shadowColorSet$insertColor(x, c)

effects: $\neg\exists\, \langle x, f \rangle \;\in this_{pre} \;\Rightarrow\; this_{post} \;=\; this_{pre} \cup \{\langle x, \text{grape} \rangle\}$

void FlavorSet.shadowColorSet$delete(x)

effects: $\exists\, \langle x, f \rangle \in this_{pre} \;\Rightarrow\; this_{post} \;=\; this_{pre} - \{\langle x, f \rangle\}$,

else throws NoSuchElementException

These definitions satisfy I and, along with the MF, satisfy property (3.13). In fact, the shadows for insert and delete have the same specifications as the real methods with the same names, i.e., the specification for ColorSet.shadowFlavorSet$insert is the same as that for ColorSet.insert, and so on. This is because ColorSet and FlavorSet both inherit the mutators insert and delete from IntSet, so it is particularly easy to define these shadow methods.

Suppose we had instead defined our MF as:

$$O_{new} \;=\; MF(O_{old}) \quad = \quad \{\, \langle x, \text{cherry} \rangle \mid x \;\in\; O_{old} \,\} \tag{3.18}$$

Now, our upgrade specification no longer satisfies property (3.13):

$$MF(O_{old}.\text{insert(x)}) \quad \neq \quad MF(O_{old}).\text{shadowFlavorSet\$insert(x)} \tag{3.19}$$

because in the first case, x has flavor "cherry," while in the second case, x has flavor "grape."

This kind of inconsistency is a problem for systems that use replication, because different replicas may introduce the new object at different times and therefore may have different flavors for the same elements. We could use synchronization to control when replicas introduce the new object, but this may stall the system when nodes or the network fail. Alternatively, we can repair inconsistencies using state transfer, but not all systems support this. Therefore, it is a good idea for upgrade specifications to satisfy property (3.13).

Had we chosen invariant (3.15), these shadow methods would not work. The shadows for insertColor and insertFlavor would need to preserve the color-flavor mapping required by (3.15). For example, O_{old}.shadowFlavorSet$insertFlavor(x, cherry) would need to add x to O_{old} with the color "red."

Had we chosen invariant (3.16), our original mapping function and shadow methods would work, but we could use even weaker ones. For example, we could define FlavorSet.shadowColorSet$delete to have no effect.

Example: Replace IntSet with CompactSet

All the methods in the previous example had deterministic effects (except failure exceptions, which are non-deterministic). We now show how to define an upgrade when methods have non-deterministic effects.

This upgrade replaces objects of class IntSet (Figure 3-1, page 33) with objects of class CompactSet, where CompactSet is the same as IntSet, except it replaces delete with compact:

void compact()

effects: $this_{post} \subseteq this_{pre}$

That is, compact removes some subset of elements from the set (possibly none). Since CompactSet does not have delete, this upgrade is incompatible.

We define I as:

$$O_{old} \quad = \quad O_{new} \tag{3.20}$$

i.e., CompactSet and IntSet always contain the same set of elements. This means our mapping function is simply the identity map.

The shadow methods for insert (in either direction) are straightforward: CompactSet.shadowIntSet$insert has the same effect as CompactSet.insert, and IntSet.shadowCompactSet$insert has the same effect as IntSet.insert.

The shadow of delete on CompactSet must remove x from the set to satisfy I:

void CompactSet.shadowIntSet$delete(x)

effects: $x \in this_{pre} \Rightarrow this_{post} = this_{pre} - \{ x \}$,

else throws NoSuchElementException

What about the shadow of compact on IntSet? I requires that when elements are removed from CompactSet, the *same* elements must be removed from IntSet. But we cannot express this just using $this_{pre}$ and the (non-existent) arguments to IntSet.shadowCompactSet$compact. To express this effect, the specification of this shadow method needs to refer to the pre and post states of the companion object, i.e., CompactSet. We refer to these states as $that_{pre}$ and $that_{post}$:

void IntSet.shadowCompactSet$compact()

effects: $this_{post} = that_{post}$

When the specification of a shadow method refers to $that_{pre}$ and $that_{post}$, the meaning is that the non-shadow method runs first, producing the post state of the companion; then the shadow runs, and it can refer to the pre and post states of the companion object.

But most shadow specifications do not need this extra information, since given $this_{pre}$ and I, we know that $that_{pre}$ satisfies $I(this_{pre}, that_{pre})$; and if the non-shadow method is deterministic, then we can deduce information about $that_{post}$ from what we know of $that_{pre}$.

It may be tempting to specify the effects clauses of all shadow methods as $I(this_{post}, that_{post})$. But if the invariant is not one-to-one, this is not precise enough. Furthermore, specifying shadow methods in terms of the invariant obscures the meaning of the specification (the implementor must consider both the non-shadow specification and the invariant together to deduce what the shadow is supposed to do). Therefore, we advocate writing out the effects clauses of shadow methods in full.

3.4.5 Disallowed Calls

In all the examples so far, we have been able define the shadow methods for an upgrade. But sometimes this isn't possible, because preserving the invariant I between two objects causes one of them to violate its *history properties*, i.e., make an illegal state transition (Section 3.1.2).

We resolve such conflicts by disallowing methods. We require:

Subtype Rule After removing disallowed methods, a type T with its shadow methods must be a behavioral subtype of T.

Our disallow constraint requires that we never disallow calls to the current type. Therefore, there are two cases to consider:

1. T_{new} is simulated, so we can disallow T_{new}'s methods but not T_{old}'s (because it may be the current type)

2. T_{old} is simulated, so we can disallow T_{old}'s methods but not T_{new}'s (because it may be the current type)

This is stricter than necessary, because when both types are simulated (i.e., neither is the current type), we could disallow calls to either one. We limit ourselves to the two cases above to simplify the process of determining which methods to disallow and simplify the implementation of simulation objects.

We want to disallow the minimal set of calls to an type so that clients cannot observe violations of the history properties of either type. Let's consider the two cases above (T_{new} is simulated, T_{old} is simulated) in turn. When T_{new} is simulated, we disallow enough T_{new} methods so that these properties hold (T^{-}_{new} is T_{new} with its disallowed methods removed):

P1 T^{-}_{new} with the T_{old} shadows is a subtype of T^{-}_{new}.

P2 When the transform function runs and O_{new} becomes the current object, clients that were using T^{-}_{new} and start using T_{new} observe no violation of T_{new}'s history properties.

P3 T_{old} with the T^{-}_{new} shadows is a subtype of T_{old}.

Therefore, the T^{-}_{new} shadows in **P3** are the shadows of all T_{new} mutators that weren't disallowed by **P1**.

When T_{old} is simulated, we have the reverse case, i.e., we disallow enough T_{old} methods so that these properties hold:

P4 T^-_{old} with the T_{new} shadows is a subtype of T^-_{old}.

P5 T_{new} with the T^-_{old} shadows is a subtype of T_{new}.

T^-_{old} is T_{old} with its disallowed methods removed.

Thus, our approach for disallowing methods is as follows. Start by assuming T_{new} is simulated, and define shadows for the T_{old} mutators (because they must be allowed). Decide which T_{new} methods must be disallowed to guarantee **P1** using induction over computations of the node that include calls to T_{new}'s allowed methods, T_{new}'s shadows of T_{old}'s methods, and the upgrade that makes O_{new} the current object. Clearly the set of disallowed methods may include observers of T_{new}, but it may also include mutators to satisfy **P2**.

Next, define shadows for the allowed T_{new} mutators. If this isn't possible for a given T_{new} mutator (because this would violate **P3**), it must also be disallowed.

Now consider the case when T_{old} is simulated. Define a shadow for each T_{new} mutator that was disallowed previously (because they must now be allowed). Then decide which T_{old} methods must be disallowed to guarantee **P4** and **P5**. Do not consider computations that include the upgrade, because it has already run at this point.

The problem with disallowing is that it cripples the objects involved and so may degrade service. An alternative is to weaken the invariant I and adjust the shadow methods so that these properties are satisfied:

$$\begin{aligned} I(O_{old}, O_{new}) \quad &\Rightarrow \quad I(O_{old}.m(args), O_{new}.shadowT_{old}\$m(args)) \\ &\qquad \wedge\; H_{new}(O_{new}, O_{new}.shadowT_{old}\$m(args)) \end{aligned} \tag{3.21}$$

$$\begin{aligned} I(O_{old}, O_{new}) \quad &\Rightarrow \quad I(O_{old}.shadowT_{new}\$p(args), O_{new}.p(args)) \\ &\qquad \wedge\; H_{old}(O_{old}, O_{old}.shadowT_{new}\$p(args)) \end{aligned} \tag{3.22}$$

H_{new} is the history property for T_{new}; it must hold for all sequences of states of O_{new}. By requiring that shadow methods obey the history properties of a type, we guarantee that our upgrade specification obeys the subtype rule.

Let's consider an example of how to apply these techniques.

Example: Replace IntSet with GrowSet

This upgrade replaces IntSet (Figure 3-1, page 33) with GrowSet (IntSet without delete). GrowSet has a history property: later states are always supersets of earlier ones. This example is analogous to upgrades that strengthen the guarantees provided by a system. For example, CFS [42] guarantees that files are stored for a certain time period, after which the system may discard them. An upgrade that extends this time period strengthens the history properties of CFS, because the new version makes all the guarantees of the old one and more. It is also possible for an upgrade to weaken history properties, e.g., by changing CFS to allow explicit deletion of files.

Returning to our example, we define our invariant I as:

$$O_{old} \quad = \quad O_{new} \tag{3.23}$$

i.e., O_{old} and O_{new} implement the same set of integers throughout the computation. Thus, our mapping function is just the identity map.

We start by assuming GrowSet is simulated, so we need to define shadows for IntSet's mutators, i.e., GrowSet.shadowIntSet$insert and GrowSet.shadowIntSet$delete. To preserve our invariant, GrowSet.shadowIntSet$insert(x) must add x to the set and GrowSet.shadowIntSet$delete(x) must remove x from the set.

These shadow definitions mean that mutations on IntSet may violate the history properties of GrowSet. If this violation is exposed to clients, those clients may break. Therefore we must disallow the methods of GrowSet that could reveal the violation. We must disallow GrowSet.contains, since it might allow clients to observe x as being in the set and later missing, e.g., GrowSet.contains(x); IntSet.delete(x); GrowSet.contains(x). We must also disallow GrowSet.insert, since clients that see insert(x) succeed will expect to see x in the set after the node upgrades, but it might have been removed, e.g., GrowSet.insert(x); IntSet.delete(x); TF; GrowSet.contains(x) (TF is the transform that makes GrowSet the current object).

Since T_{new}^{-} has no methods, **P1–P3** are trivially satisfied. But if we were to allow either GrowSet method, **P1** or **P2** would be violated.

Next, we consider the case when IntSet is simulated. Now all of GrowSet's methods are allowed, and we must define shadows for any of them that are mutators. There is just one shadow, IntSet.shadowGrowSet$insert, and it has the same effect as IntSet.insert. Finally, we need to determine which methods of IntSet must be disallowed. We do this by considering whether the shadows of GrowSet's mutators violate IntSet's history properties (it has none, so they don't, and **P4** is trivially satisfied) and whether the shadows of IntSet's methods violate GrowSet's history properties. GrowSet.shadowIntSet$insert is fine, but GrowSet.shadowIntSet$delete is no good (it would violate **P5**). Therefore, we disallow IntSet.delete.

In this example, we had to disallow methods of both simulated types, and in the case of GrowSet, we had to disallow all of its methods. We would like to avoid having to disallow so many methods, and one way to do this is to weaken our invariant. If we define I as:

$$O_{old} \quad \subseteq \quad O_{new} \tag{3.24}$$

we can specify that GrowSet.shadowIntSet$delete(x) has no effect. Then elements need not be removed from GrowSet, and we need not disallow any methods. But this approach violates property (3.13):

$$MF(O_{old}.\mathsf{delete(x)}) \quad \neq \quad MF(O_{old}).\mathsf{shadowIntSet\$delete(x)} \tag{3.25}$$

In the first case, x is not in the GrowSet, while in the second case, x is in the GrowSet.

If we keep our original invariant, then we have to accept the fact that we cannot simulate GrowSet while IntSet is the current object, and we cannot simulate IntSet.delete while GrowSet is the current object. We can plan our upgrade to minimize the impact of this limitation, e.g., we could use a schedule that upgrades IntSet nodes before upgrading the clients of those nodes, so that no clients attempt to use GrowSet while it is being simulated.

3.5 Example

In Section 2.5, we presented an upgrade from PermServer, a file system that supports Unix-style permissions, to AclServer, a file system that supports access control lists. In this section, we define this upgrade using the methodology presented in this chapter.

This upgrade is incompatible: PermServer and AclServer are unrelated by subtyping. Therefore, we will need to define an invariant, mapping function, and shadow methods.

3.5.1 Invariant

The first step in defining this upgrade is to define an invariant I between the abstract states of the two types.

The abstract state, P, of PermServer is a mapping from files to owner names, group names, and permission bits:

$$\mathsf{P} = \mathsf{filename} \rightarrow \langle\ \mathsf{ownername, groupname, Or, Ow, Ox, Gr, Gw, Gx, Wr, Ww, Wx}\ \rangle$$

where O, G, and W refer to the file owner, file group, and the rest of the world, respectively; and r, w, and x refer to read, write, and execute permission, respectively.

The abstract state, A, of AclServer is a mapping from files to sets of access rights:

$$\mathsf{A} = \mathsf{filename} \rightarrow \langle\ \mathsf{name, r, w, x, a}\ \rangle^*$$

where name is the name of a user or group; and r, w, x, and a refer to read, write, execute, and modify-ACL permission, respectively.

Ideally, we would define an invariant that guarantees that users have the same access rights whether they use PermServer or AclServer. But this is impossible, because while a set of permissions can always be expressed as an ACL, an ACL cannot always be expressed as a set of permissions. Therefore, the invariant must be *lossy*, i.e., it may need to throw away information when mapping an AclServer to a PermServer. Our goal is to define an invariant that throws away as little information as possible.

We define an invariant I between P and A as a per-file, bidirectional mapping:

$$
\begin{aligned}
&\mathsf{P(filename)} = \langle\ \mathsf{ownername, groupname, Or, Ow, Ox, Gr, Gw, Gx, Wr, Ww, Wx}\ \rangle\\
&\qquad\qquad \Leftrightarrow\\
&(\ \langle\ \mathsf{ownername, Or, Ow, Ox, true}\ \rangle \in \mathsf{A(filename)}\\
&\qquad\qquad \vee\ (\mathsf{ownername} = \text{“nobody”}, \mathsf{Or = false, Ox = false, Ow = false})),\\
&(\ \langle\ \mathsf{groupname, Gr, Gw, Gx, false}\ \rangle \in \mathsf{A(filename)}\\
&\qquad\qquad \vee\ (\mathsf{groupname} = \text{“nobody”}, \mathsf{Gr = false, Gx = false, Gw = false})),\\
&(\ \langle\ \mathsf{sys{:}anyuser, Wr, Ww, Wx, false}\ \rangle \in \mathsf{A(filename)}\\
&\qquad\qquad \vee\ (\mathsf{Wr = false, Wx = false, Ww = false}))
\end{aligned}
$$

This invariant says that for each set of permissions in P, there is a corresponding access control list in A that contains the same set of access rights for the file's owner, the file's group, and the rest of the world. The ACL may contain additional rights for other users or groups as well.

The invariant also says that for each access control list in A, there is a corresponding set of permissions in P. To handle the problem that ACLs may have information that cannot be expressed as a set of permissions, *I* allows permissions to be set for "nobody," a special user/group that has no permissions. This definition of *I* has ambiguity: it allows for permissions to be set for "nobody" even if the ACL has entries that could be mapped to specific users and groups. We could strengthen *I* to prevent this; but instead, we'll use the shadow methods to resolve the ambiguity.

3.5.2 Mapping Function

The mapping function *MF* defines the initial state A of AclServer given a state P of PermServer. We define *MF* as follows:
For each file f in P,
where P(f) = ⟨ ownername, groupname, Or, Ow, Ox, Gr, Gw, Gx, Wr, Ww, Wx ⟩:

- Set the ACL for file f in A to:
  `user:`owner: Or Ow Ox `true`
  `group:`group: Gr Gw Gx `false`
  `system:anyuser:` Wr Ww Wx `false`

where each line in the ACL is a name-rights mapping in A: the first element in each line is the type of principal—user, group, or system (special); the second element is the name of the principal; and the third element is the set of rights for that principal (expressed as four `true` or `false` bits).

MF establishes *I*: the initial ACL for each file is one of the ACLs allowed by *I* for the given permissions of f.

3.5.3 Shadow Methods

The next step in defining this upgrade is to specify its shadow methods. Let's start with the shadows of PermServer's methods. There is only one mutator, setPerms:

AclServer.shadowPermServer$setPerms(f, owner, group, bits)

- Sets the ACL for file f to:
  `user:`owner: Or Ow Ox `true`
  `group:`group: Gr Gw Gx `false`
  `system:anyuser:` Wr Ww Wx `false`

This specification preserves I, because this ACL is one of the ACLs allowed by I for the given permissions of f. This specification means the new ACL may throw away access rights for principals that were specified on the old ACL. If we want to avoid this, we could instead specify that this shadow just adds lines to the ACL rather than replacing the ACL.

Now we specify the shadows of AclServer's methods. Again, there is just one mutator, setACL:

PermServer.shadowAclServer$setACL(f, acl)

- Sets f's owner to the first user in acl with the "a" permission, or "nobody" if no such user exists.
- Sets f's group to the first group in acl with no "a" permission, or "nobody" if no group exists.
- Sets the Or, Ow, Ox bits of f according to the access rights of the first user in acl with the "a" permission, or all false if no such user exists.
- Sets the Gr, Gw, Gx bits of f according to the access rights of the first group in acl with no "a" permission, or all false if no such group exists.
- Sets the Wr, Ww, Wx bits of f according to access rights of sys:anyuser in acl (if it has no "a" permission), or all false if no such rights exist.

This specification preserves I, because this set of permissions is one of those allowed by I for the given acl. But this does not necessarily create the maximally-permissive set of permissions, because, e.g., the second or third user in the ACL might have been a better choice (depending on the ACL). But this specification does better than the minimally-permissive one, which just sets the owner and group for the file to "nobody" for all ACLs.

3.5.4 Implementation Considerations

Because ACLs may contain more information than permissions, the AclServer implementation may need to keep more state than the PermServer implementation. In particular, when AclServer is implemented using a simulation object, that SO will need to keep persistent state to record the ACL information that is not stored (as permissions) in the underlying PermServer. Furthermore, when the node upgrades, and AclServer becomes the current type, the transform function will need to incorporate the SO's state when producing the state of the AclServer.

Managing the persistent state of the AclServer SO is a lot of work. If it is not necessary to provide full AclServer support, the upgrader may instead *choose* to disallow some AclServer calls so as to simplify the SO and TF implementations. In particular, the upgrader could disallow all calls to setACL(f, acl) in which acl cannot be expressed perfectly as a set of permissions. By doing so, the upgrader allows the SO to be stateless and so simplifies the TF (since it no longer needs to incorporate the SO's state). Of course, once AclServer is the current type, all setACL calls must be allowed.

Thus, disallowing-by-choice can be a useful way to manage implementation complexity. In the Chapter 4, we will discuss several other implementation-related reasons why calls may need to be disallowed.

3.6 Realizing the Sequence Requirement

The sequence requirement implies that a method call to one object (i.e., that implements one of a node's types) must reflect the effects of all earlier calls to all the node's objects (i.e., for all of its types).

To understand the behavior of an object O of type T, we need a computation history on O consisting of calls only to T's methods. However, the real computation history of a node is some arbitrary interleaving of calls to all of a node's types. We use shadow methods to rewrite this history so that it is all in terms of T.

Doing this rewriting requires that shadow methods are *expressible* in terms of the type on which they are defined:

Expressibility A shadow method on T is *expressible* if it can be defined as a sequence of calls to T's normal (non-shadow) methods.

But not all shadows are expressible, for two reasons: First, T may be missing some methods. For example, CompactSet has no delete method (it just has compact), so there is no way to express the effects of shadowIntSet$delete in terms of calls to CompactSet's methods. The ramifications of missing methods are serious, but this problem has an easy solution: the user can simply add the missing methods to T (i.e., add delete to CompactSet). Of course, this isn't reasonable when the purpose of the upgrade is to remove the methods that are now missing! Furthermore, adding the needed methods only works for new types; the user cannot add new methods to old types.

The second reason a shadow may be inexpressible is because its behavior is incompatible, e.g., shadowIntSet$delete on GrowSet. In this case the user cannot compensate by adding methods (e.g., by adding delete to GrowSet), because doing so would compromise the type's history properties.

Expressibility is not the same as *implementability*: a shadow method may be expressible but not implementable, because, e.g., the implementation would require access to state that cannot be accessed via T's methods. For example, suppose FlavorSet has a shadow whose effect is to remove all elements from the set of a particular flavor. This shadow method is expressible as a sequence of calls to FlavorSet.delete; but it is not implementable, because FlavorSet provides no way to iterate over its elements or otherwise find all elements of a particular color. This kind of problem is easy to fix by adding methods. And even if a shadow is unimplementable, we can still reason about the sequence requirement as long as the shadow is expressible.

Assuming expressibility, rewriting the computation history in terms of T's methods is easy. Start with the original computation, and get rid of all calls to observers. For each call to a mutator for a type other than T, replace it with the sequence of calls that expresses its shadow (in the direction of T). Repeat this until only calls to T are left. At this point, we know the state of O from its specification, and therefore we understand how its state evolved throughout the computation despite interleaving.

For example, if T is T_1 and the computation contains a call $T_3.m(args)$, replace this call with the effects of $T_2.shadowT_3\$m(args)$ expressed in terms of T_2's methods, e.g., $T_2.p(args); T_2.q()$. Next, replace each of these calls with the effects of $T_1.shadowT_2\$p(args)$ and $T_1.shadowT_2\$q()$, expressed in terms of T_1's methods. The result is a computation expressed purely in terms of T_1's methods that allows us to understand how O_1 evolved throughout the computation.

Inexpressibility doesn't cause difficulties in reasoning about the sequence requirement for the two types involved in an upgrade (e.g., IntSet and GrowSet), because we have the shadow method specifications. For example, the specification for shadowIntSet$delete defines what happens to a GrowSet object when IntSet.delete is called. (Of course, in this case, the GrowSet object is unable to do much anyway when it's being simulated.) But inexpressibility does cause problems when reasoning about the other (older and newer) types on the node. The problem is that to take the next step in the rewriting described above, we need to be able to interpret the effects of the shadow method on the other types. So far, we have no way of doing this.

There are two cases to consider. First, a new type may have mutators whose shadows are inexpressible as methods of the old type. In this case, we're constrained: we cannot add methods to the old type to make the shadows expressible, nor can we change the definitions of previous types to define the effects of the inexpressible shadows. Our only option is to rule out (disallow) calls that would prevent us from reasoning about the sequence requirement. So that the upgrade system can disallow the appropriate calls, the upgrader must designate the methods of the new type whose shadows are inexpressible (as part of defining the upgrade).

The second case is that an old type may have mutators whose shadows are inexpressible as methods of the new type. In this case, we have three options: First, we could make the shadows expressible by adding methods to the new type. Second, we could disallow calls that would prevent us from reasoning about the sequence requirement; in this case, the upgrader must designate the old methods whose shadows are inexpressible (as part of defining the upgrade). Third, we could specify the effects of the inexpressible shadows on later (newer) types directly.

We discuss these two cases in turn.

3.6.1 Inexpressible New Methods

Suppose we have a sequence of two upgrades, $T_{oldold} \rightarrow T_{old}$ and $T_{old} \rightarrow T_{new}$, and the shadow of $T_{new}.m$ is inexpressible on T_{old}. In this case, we cannot add methods to T_{old} to make $T_{old}.shadowT_{new}\$m$ expressible; and there is no way to define the effects of $T_{old}.shadowT_{new}\$m$ on T_{oldold}, as we have no way of providing an interpretation of a shadow that didn't exist at the time T_{oldold} was defined.

To guarantee that we can always rewrite the computation history into methods of any type, our only options are to disallow calls to $T_{new}.m$ or to disallow T_{oldold} (and all earlier types) entirely. Because of our disallow constraint, we cannot disallow $T_{new}.m$ while T_{new} is the current type, and we cannot disallow calls to T_{oldold} while it is the current type. When neither T_{new} nor T_{oldold} is the current type, we could take either approach. Furthermore, once T_{oldold} is retired, we can allow $T_{new}.m$ (assuming it's not disallowed for other reasons).

Our approach is as follows. First, the upgrader designates $T_{new}.m$ as inexpressible as part of defining the $T_{old} \rightarrow T_{new}$ upgrade. While T_{oldold} (or some older type) is the current type, all calls to $T_{new}.m$ are disallowed (this often happens anyway because of incompatibilities).

For example, suppose T_{oldold} is ColoredCompactSet (a subtype of CompactSet), T_{old} is CompactSet, and T_{new} is IntSet. The upgrader designates IntSet.delete as inexpressible, because there is no sequence of calls to CompactSet that can implement shadowIntSet$delete. While Colored-CompactSet is the current type, all calls to IntSet.delete are disallowed.

Next, when the node upgrades and T_{old} becomes the current type, we have a choice: we can either continue to disallow $T_{new}.m$, or we could disallow *all* calls to T_{oldold} (and all earlier types). We choose the former, as we expect it to be less disruptive (although the latter may be more appropriate for certain upgrades). For example, when CompactSet becomes the current type, all calls to Int-Set.delete are still disallowed.

When the node upgrades again and T_{new} becomes the current type, we disallow *all* calls to any type older than T_{old}, i.e., to T_{oldold} and all earlier types. (The disallowing is actually implemented by the upgrade system, since the implementation of T_{oldold} does not know that calls need to be disallowed.) This gives us the sequence requirement trivially: since there are no calls to the T_{oldold} object, we do not need to define the effects $T_{new}.m$ on it. We can never allow calls to T_{oldold} again after this point, since there is no way to know what state it should have after $T_{new}.m$ is called.

For example, when the node upgrades and IntSet becomes the current type, all calls to Colored-CompactSet (and all earlier types) are disallowed. We can never allow calls to ColoredCompactSet again after this point.

3.6.2 Inexpressible Old Methods

Suppose we have a sequence of two upgrades, $T_{old} \rightarrow T_{new}$ and $T_{new} \rightarrow T_{newnew}$, and the shadow of $T_{old}.m$ is inexpressible on T_{new}. In this case, we have three options. First, we could add methods to T_{new} to make $T_{old}.m$ expressible. But suppose T_{old} is IntSet, T_{new} is GrowSet, T_{newnew} is ColoredGrowSet (a subtype of GrowSet), and $T_{old}.m$ is IntSet.delete. In this case, it is inappropriate to add delete to GrowSet.

Our second option is to disallow methods: we can disallow calls to $T_{old}.m$ or disallow T_{newnew} (and all later types) entirely. Because of our disallow constraint, we cannot disallow $T_{old}.m$ while T_{old} is the current type, and we cannot disallow calls to T_{newnew} while it is the current type. We will discuss this option in detail in this section.

Our third option is to specify the effects of $T_{new}.shadowT_{old}\$m$ on T_{newnew} directly, i.e., we can define a shadow of the shadow method. We discuss this option in the next section.

If we choose to disallow methods to address this problem, the upgrader designates $T_{old}.m$ as inexpressible as part of defining the $T_{old} \rightarrow T_{new}$ upgrade. For example, the upgrader designates IntSet.delete as inexpressible as part of defining the IntSet → GrowSet upgrade.

While T_{old} (or any older type) is the current type, we disallow *all* calls to any type newer than T_{new}, i.e., to T_{newnew} and all later types. For example, while IntSet is the current type, all calls to ColoredGrowSet are disallowed.

When the node upgrades and T_{new} becomes the current type, we have a choice: we could either continue to disallow calls to T_{newnew}, or we could disallow all calls to $T_{old}.m$. We choose the latter, as it seems less disruptive to disallow one method than a whole type, and $T_{old}.m$ must often be disallowed anyway due to incompatibilities. This gives us the sequence requirement, because the initial state of T_{newnew}'s object is defined by the mapping function at the moment T_{new} becomes the current type, and all later calls can be rewritten in terms of T_{newnew}'s methods.

For example, when the node upgrades and GrowSet becomes the current type, all calls to ColoredGrowSet are allowed, and all calls to IntSet.delete are disallowed (this happens anyway for the reasons described in Section 3.4.5). The initial state of ColoredGrowSet is the result of applying the mapping function to GrowSet at the moment GrowSet becomes the current type.

Finally, when the node upgrades and T_{newnew} becomes the current type, we continue to disallow calls to $T_{old}.m$. We continue to disallow calls to $T_{old}.m$ even for later upgrades, as at that point $T_{old}.m$ is several versions in the past (and T_{old} is probably retired).

3.6.3 Shadows of Shadows

Instead of disallowing calls to T_{newnew} when $T_{old}.m$ is inexpressible, we can simply specify the effect of $T_{new}.shadowT_{old}\$m$ on T_{newnew}: this is just a shadow's shadow. Adding this extra specification means we can allow calls to T_{newnew}, because we can now explain the effects of calls to T_{old} on T_{newnew} and so satisfy the sequence requirement.

For example, suppose T_{old} is IntSet, T_{new} is CompactSet, and T_{newnew} is ColoredCompactSet. The shadow of IntSet.delete on CompactSet, shadowIntSet$delete, is inexpressible. When the upgrader defines the CompactSet → ColoredCompactSet upgrade, this upgrade can include a specification for the shadow's shadow:

void ColoredCompactSet.shadowCompactSet$shadowIntSet$delete(x)

effects: $\exists\, \langle x, c\rangle \in this_{pre} \;\Rightarrow\; this_{post} \;=\; this_{pre} - \{\langle x, c\rangle\}$,

else throws NoSuchElementException

Like any other shadow method, this must preserve the invariant between CompactSet and ColoredCompactSet. This specification allows us to explain the effects of a call to IntSet.delete on a

ColoredCompactSet object; therefore we can allow calls to all three types simultaneously. In this case, the shadow's shadow is inexpressible, so the next type after T_{newnew} will need to specify a shadow for the shadow's shadow. If T_{newnew} were instead ColorSet, the shadow's shadow would be expressible, and there would be no need to define a shadow for the shadow's shadow.

Unfortunately, a shadow's shadow might be incompatible with the type on which it is defined! Suppose T_{old} is IntSet, T_{new} is GrowSet, and T_{newnew} is ColoredGrowSet. The invariant between GrowSet and ColoredGrowSet requires that they contain the same set of elements, and since GrowSet.shadowIntSet$delete(x) removes x from GrowSet, ColoredGrowSet.shadowGrowSet$shadowIntSet$delete(x) must also remove x from ColoredGrowSet. This violates ColoredGrowSet's history properties, which means it must disallow methods as described in Section 3.4.5.

But it is not acceptable to simply disallow ColoredGrowSet's methods as usual, as this would mean they were disallowed even when the node upgrades and GrowSet is the current type! In this case, we want to allow all of ColoredGrowSet's methods. This means we need a way to distinguish methods disallowed because of shadows from those disallowed because of shadows' shadows (and so on).

We propose the following approach. Methods that are disallowed because of shadows (e.g., all of GrowSet's methods, because of shadowIntSet$delete) are simply marked as "disallowed" in the upgrade definition. Calls to these methods are disallowed until the type on which they are defined (i.e., GrowSet) becomes the current type. Methods that are disallowed because of shadows' shadows (e.g., all of ColoredGrowSet's methods, because of shadowGrowSet$shadowIntSet$delete) are marked as "shadow-disallowed" in the upgrade definition. Calls to these methods are disallowed until the type *previous to* the type on which they are defined becomes the current type (again, this is GrowSet). This procedure can continue ad infinitum, but it is unlikely to be necessary beyond this level.

The set of methods that are disallowed for a type gets smaller as that type gets closer to becoming the current type. A type T_{new} could have shadow-shadow-disallowed methods, shadow-disallowed methods, and disallowed methods; these would be disallowed until T_{oldold}, T_{old}, and T_{new} become the current type, respectively. When T_{newnew} becomes the current type, some T_{new} methods might again be disallowed, but this would be because of conflicts with T_{newnew} only, not any earlier types.

Chapter 4

Simulation Objects

Chapter 3 presented our abstract model for how nodes implement multiple types simultaneously. This chapter presents several designs for realizing this model using simulation objects.

We begin with a discussion of previous approaches to implementing multiple types on a single node. Some of these approaches fail to meet the requirements we set forth in the previous chapter; others meet the requirements but have poor expressive power (i.e., calls must often be disallowed); still others provide good expressive power but are impractical to implement for more than a few types. We explain why our approach is practical and is more powerful than previous approaches.

We then present various ways to use simulation objects to implement multiple types. These models differ in how calls are dispatched to objects (i.e., which objects implement which types) and how simulation objects can interact with one another. Different models offer different tradeoffs between ease of implementation and expressive power.

The first model is the *interceptor model*. In this model, the simulation object for the latest version handles all calls (it *intercepts* calls intended for the earlier versions). It can delegate calls to the previous type, which may be implemented by the current object or another SO. The interceptor model is simple and powerful, because a single object manages all the types of the node. But for the same reason, this model is difficult to use when types are incompatible.

The second model is the *direct model*. In this model, calls for each version are dispatched *directly* to the object that implements the type for that version, which the current object or a simulation object. Each SO implements just its own type and can delegate calls to the next object closer to the current object. This means the direct model is practical regardless of how many types there are and how they are related. However, this model has limited expressive power (i.e., calls must often be disallowed), because it is difficult to ensure that the effects of a call are reflected on all versions.

We can combine the first two models in a *hybrid model* that provides the benefits of both models. As there are several ways to combine the direct and interceptor models, there are several variants of the hybrid model that offer different tradeoffs.

However, even the hybrid model and its variants have their weaknesses, so we present another approach: the *notification model*. This model is like the direct model in that calls for a version are dispatched to the object for that version. However, instead of having each SO delegate to a single other object, we have each SO *notify* the objects for the next and previous versions on each method call. Those objects respond to the notification by updating their state (as specified by the shadow of the method that the notifier received); then they propagate the notification to the next objects. The notification model has more expressive power than the other models, but it also requires more work from the upgrader.

After describing the various models, we discuss how well each supports concurrency control. This is a vital concern in distributed systems, since nodes typically handle many clients in parallel. We choose not to serialize calls in the upgrade layer, because this would cause unacceptable performance degradation and, in some cases, could cause deadlock. Instead, we require that simulation objects handle concurrency themselves. Some of the simulation models can handle concurrency well, but others must rely on application-level concurrency control.

We conclude with a discussion of the tradeoffs between the different models, guidelines for how to choose the right model for a given system, and a summary of the reasons why calls may be disallowed.

4.1 Previous Approaches

When different nodes or objects in a system run classes from different schema, we say the system is running in *mixed mode*. Our approach relies on mixed mode to allow upgrades to be scheduled, and other systems have used similar approaches, both in distributed systems and in other domains. In this section, we compare several techniques for supporting mixed mode.

The basic idea behind existing techniques for supporting mixed mode is to allow each node in the system to implement multiple types simultaneously—one for each version of the system. When one node makes a method call on another, the caller assumes that the callee implements a particular type. In reality, the assumed type may be just one of several that the callee implements. This design simplifies software development by allowing implementors to write their software as if every node in the system were running the same version.

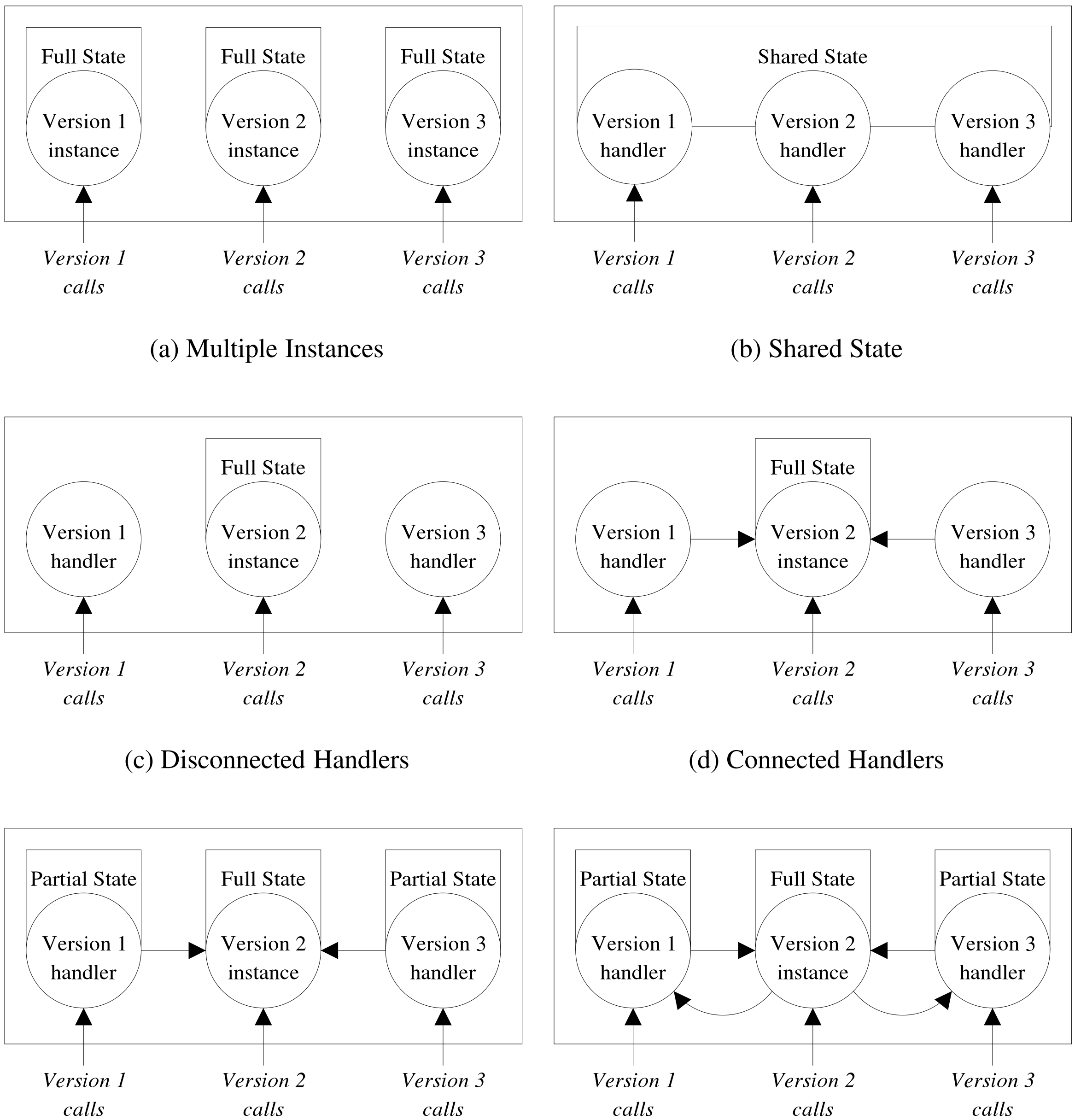
The simplest way that a node could implement multiple types is by running instances of each version side-by-side, each with its own state, as depicted in Figure 4-1(a). For example, a node might implement two versions of a file system specification by running instances of both versions side-by-side. A caller that interacts with only one of the two instances can store and retrieve files as usual. Two callers that interact with the same instance can share files. But if two callers are running different versions, they interact with different instances and cannot share files. A single caller may lose access to its own files by storing files at one version, then upgrading, then attempting to fetch files at the next version. Since each instance of the file system has its own state, the files that the caller stored at one version are inaccessible at the next.

The problem with the multiple-instances approach is that calls to one instance are not reflected on the states of the other instances. To allow calls to one type to be reflected on the others, the implementations of those types must share a single copy of the node's state. A straightforward way to do this is to allow them to share state directly, as illustrated in Figure 4-1(b). Unfortunately, this is just too complex. The implementations of each type must somehow avoid changing the shared state in such a way that would break the implementations of the other types. This is non-modular: implementing each additional type becomes increasingly difficult.

To ensure that a node provides at least some useful functionality, one type can be designated the *current* type for the node. The node runs an instance of the current type and so can support calls to that type perfectly. To support other types, the node runs *handlers*.

The simplest handler-based model is illustrated in Figure 4-1(c) and is similar to Skarra and Zdonik's schema versioning model for OODBs [97]. Calls to the current type are dispatched to an instance, and calls to other types are dispatched to stateless error handlers that can substitute results for those calls. This model works only if the error handlers can return a sensible default for the calls they implement. This model is also limited in two ways: first, handlers cannot implement behaviors that use (observe or mutate) the current instance's state, and second, handlers cannot implement behaviors that use state outside of that instance.

One can address the first limitation by allowing handlers to access the current instance's state via its methods, as illustrated in Figure 4-1(d). This model lets handlers share state safely and lets



(a) Multiple Instances

(b) Shared State

(c) Disconnected Handlers

(d) Connected Handlers

(e) Simulation Objects (Direct Model)

(f) Simulation Objects (Notification Model)

Figure 4-1: *Systems for supporting multiple types on a single node. Each node (large box) supports calls at versions 1, 2, and 3. Arrows indicate the direction of method calls. In (a), the node runs instances of each type. In (b), the node runs handlers for each type that share state directly. In (c), (d), (e), and (f) the node runs an instance of version 2's type and has handlers for versions 1 and 3.*

them support behaviors that can be defined in terms of the instance's behavior. Examples of this approach include the *interprocedures* of PODUS [48] and the *wrappers* of the Eternal system [102].

The problem with this model is that it is only practical for a small number of types. To enable a node to support calls on N types, one must define $N - 1$ handlers, each of which delegates to the current type directly. These handlers must be redefined each time the current type changes (due to upgrades), which becomes impractical as the number of versions increases.

One can keep the number of handlers manageable using *handler chaining*: each type has just two handlers defined for it, one that calls methods of the next higher version and another that calls methods of the next lower version. Thus, a chain of handlers can map a call on any type to calls on the current type. Instances of the handler chaining model include Monk and Somerville's update/backdate model for schema versions in OODBs [81] and Senivongse's "evolution transparency" model for distributed services [93].

The problem with handler chaining is that it may prevent handlers from implementing certain behaviors: e.g., if versions 1 and 3 support a state-accessing behavior that version 2 does not, then a version 1 handler cannot implement that behavior since it cannot call version 3 directly. This illustrates a general design tradeoff: by incorporating knowledge of additional types (thus, additional complexity), handlers may be able to better implement their own type.

None of these previous models address the second limitation mentioned above: they do not allow handlers to implement stateful behaviors that cannot be defined in terms of the current type. Our solution addresses this limitation by allowing handlers—that we call *simulation objects*, or *SOs*—to implement calls both by accessing the state of the instance via its methods and by accessing *their own* state, as illustrated in Figure 4-1(e).

Simulation objects can implement more behaviors than stateless handlers, and unlike the multiple-instances approach (Figure 4-1(a)), simulation objects can ensure that calls to one type are reflected on the others. But using handler chaining can still require that calls be disallowed, as we discuss in Section 4.3.1. We can do better be allowing information about method calls to flow in both directions, as illustrated in Figure 4-1(f). We discuss this model in Section 4.5.

We now present our models for how to use simulation objects.

4.2 Interceptor Model

When a node hears of an upgrade that affects it (i.e., the upgrade includes a class upgrade whose old class is the node's current class), it immediately installs a future SO for the new class. In the interceptor model, this SO takes over: it receives all calls intended either for the previous object or for itself. The SO implements all calls for both objects; it may do so using its own state or by calling methods of (i.e., delegating to) the previous object. We call this the interceptor model because the SO *intercepts* calls intended for its delegate, rather than letting the delegate handle them directly.

When the node upgrades, it replaces its current object and the future SO with an instance of the new class; this instance becomes the current object of the node. There is no need for a past SO, because calls made by clients running at the old version are handled by the current object. This means the current object must implement both the old and new types.

There could be a new upgrade that comes along before the node has upgraded to the new class. In this case, the node installs another SO that intercepts calls for all previous objects; the SO may delegate to the immediately preceding object. When the node upgrades, it replaces its current object and its oldest SO with an instance of the new class for its oldest pending upgrade. Figure 4-2 depicts this process.

We optimize in the case where the new class implements the same type as the old class. In this case, the node just delegates all calls for the new type to the old one, and we don't need an SO.

4.2.1 Discussion

The interceptor model works well for compatible upgrades, because the new type (implemented by the interceptor) is always a subtype of the old type. But this is not true of incompatible upgrades, using the interceptor model for them is more difficult.

In an incompatible upgrade, the new type is either a supertype of the old type or it is unrelated. An interceptor must implement the new type as well as the old one. (Incompatible interceptors need a way to distinguish calls intended for the old type from those for the new type, since there may be name conflicts. This is an implementation detail that we discuss in Chapter 7.) In the unrelated-type case, the SO cannot simply delegate the old methods, because calls to the old type may affect the new object (as specified by the shadows of the old methods on the new type). When the node upgrades and replaces the SO with an instance of the new class, this object must also implement both types; this is undesirable, because it means the new object has to implement legacy behavior.

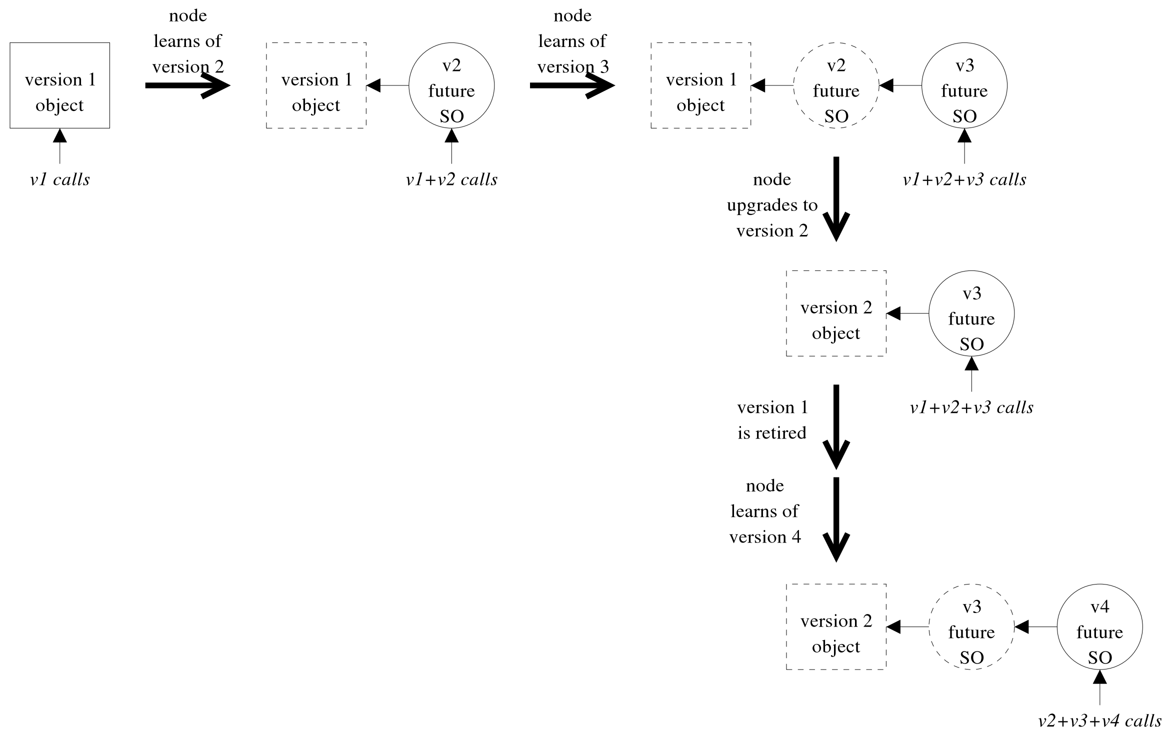


Figure 4-2: *The interceptor model, presented as a sequence of states of a node. Large arrows are state transitions. In each state, the box is the current object, and the circles are SOs. The newest object (solid border) handles all calls for the node; the dotted-border objects are "hidden" by the interceptor. Objects may delegate calls as indicated by the small arrows.*

While undesirable, supporting two types in a single object is reasonable and occurs in some real systems, e.g., NFS servers typically implement both the NFSv2 and NFSv3 protocols [36]. But this becomes unreasonable when we consider the next upgrade: its object must implement its own type as well as those for *both* previous versions. For example, if a node upgrades from NFSv2 to NFSv3 (unrelated types) and then from NFSv3 to NFSv4 (unrelated types), then the latest object must support all three protocols simultaneously. This is non-trivial: for example, NFSv2 and NFSv3 file accesses need to be serialized against locks acquired via NFSv4.

Using interceptors for incompatible upgrades means the upgrader must understand every type that the node supports and the relationships between them. As a node supports more types at once and as the relationships between the types become more complicated, the likelihood that the interceptor code is correct declines.

There is another reason why interceptors cannot support incompatible upgrades well: convergent upgrades. These are upgrades that replace two different classes with the same new class. For example, suppose version 1 has classes A and B, and version 2 has just class C. This means the version 2 upgrade has two class upgrades, one that replaces A with C and another that replaces B with C. Now consider a version 3 upgrade that replaces C with D. Its interceptor—which must handle calls for all three versions—cannot know whether calls for version 1 expect A's type or B's type! We could fix this by having nodes identify the type they expect in each call, but this means the version 3 interceptor needs to have code to handle both types for version 1, which makes implementing it even more difficult.

Thankfully, our other simulation models handle the problem of convergent upgrades more gracefully than the interceptor model: they keep a record of what types a node had in earlier versions in the form of past SOs.

4.2.2 Correctness

It is easy to satisfy the requirements put forth in Chapter 3 in the interceptor model, because all calls go through a single object, so it can serialize all calls to the node and can reflect the effects of each method on each version appropriately. As we will discuss in Section 4.6, this makes the interceptor model particularly attractive for applications that require concurrency control.

Sometimes we might want to disallow calls because implementing them in an SO is too inefficient, e.g., because they require expensive operations to maintain the appropriate state in the interceptor. This is okay unless those calls are part of the current type, in which case the disallow constraint requires that those calls are supported.

4.3 Direct Model

We could use interceptors for incompatible upgrades, but doing so has poor modularity as the node supports more and more types. We want modular reasoning, i.e., the upgrader should only need to know about the new version and the previous one, regardless of the number of types that a node supports. Therefore, we use a different approach for incompatible upgrades: the SOs aren't interceptors. Instead, each object receives calls only from clients running at the same version as the object, so each object only implements its own type.

SOs may delegate to the next object in the chain: the next older object for future SOs, the next newer object for past SOs. A class upgrade defines a future SO that implements the new type and a past SO that implements the old type, so the upgrader only needs to understand these two types to define both SOs. Thus, the direct model meets our modularity goals.

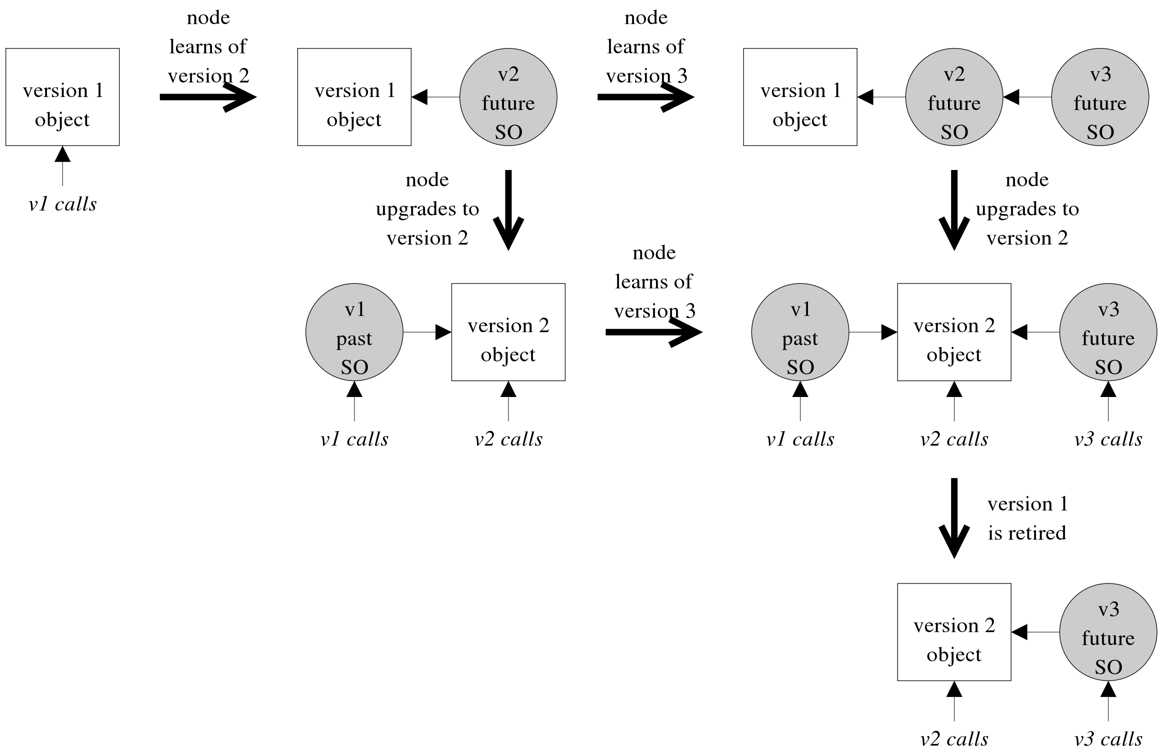


Figure 4-3: *The direct model. Each object handles calls only for its own version. Objects may delegate calls as indicated by the small arrows. Two sequences of events are possible, depending on whether the node upgrades to version 2 before or after installing the SO for version 3. Regardless of the sequence, the node always supports all the non-retired versions it knows about.*

Figure 4-3 depicts how SOs are managed in the direct model.

We optimize in the case when the new type is a supertype of the old type. Since the old type can handle all calls intended for the new type, we don't need the future SO and can simply forward the new calls to the old type. But we still require a past SO when the node upgrades, because we don't want the current object to have to provide the legacy behavior.

4.3.1 Expressive Power

We require that the effects of a method call on any of a node's types reflect the effects of all earlier calls to all of that node's types. This is simple for interceptors: they see all the calls, so they can apply the effects of each method (or shadow method) on each type. But in the direct model, an SO does not get to execute a shadow method when calls go directly to its delegate. Instead, the SO must meet the sequence requirement by calling methods of its delegate. Calls to an SO observer must reflect the abstract state of the SO, which in turn must reflect the effects of previous mutations to its delegate as defined by the shadow methods for those mutators. Similarly, calls to an SO mutator must affect the abstract state of the delegate as defined by the shadow method for that mutator.

Unfortunately, it is not always possible to implement an SO that meets our requirements in the direct model. There are two fundamental problems: First, calls that go directly to the delegate and bypass the SO may invalidate the state of the SO. Second, the effects of an SO method may be *unimplementable* by making calls to the delegate. In both cases, we must disallow some calls to the SO to meet our requirements. We discuss the two problems in turn.

The first problem with the direct model has to do with interleaving. This means an SO may not be able to tell what calls the delegate receives and so will not be able to reflect them correctly. For example, consider an SO that implements ColorSet by delegating to an object that implements IntSet. The delegate stores the state of the set (the integers in the set), and the SO stores the color for each integer in the set (the delegate knows nothing of colors). We can implement the state of the SO as a mapping from integers to colors, update this mapping upon calls to SO.insert, SO.insertColor, and SO.delete, and query this mapping on SO.getColor.

But this doesn't work, because our shadow methods require that mutations of the delegate change the state of the SO in ways that affect this mapping. Let O refer to the SO's delegate, and consider the sequence of calls SO.insertColor(1, red); O.delete(1); O.insert(1); SO.getColor(1). The result of the final call will be "red," because the SO cannot know that 1 was ever removed; but because 1 was removed and re-inserted, its color should be the one specified by ColorSet.insert(x), which is blue (see Section 3.4.2 and Figure 3-2).

The problem here is that when O.delete(1) runs, the SO does not get a chance to apply the effects of SO.shadowIntSet$delete(1). This results in a violation of an invariant between the state of the SO and the state of the delegate:[1]

$$\langle x, c \rangle \in \mathit{SO.colorMap} \quad \Rightarrow \quad x \in O$$

where *SO.colorMap* is the private state of the SO that stores the mapping from integers to colors. We might hope to correct *SO.colorMap* when the next SO method is called, but this doesn't work as demonstrated in our example: mutations to the delegate may erase the evidence that the SO state has become stale.

[1] We can think of this invariant as a *representation invariant* of the SO. The fact that calls can mutate the delegate directly is a form of *representation exposure*. Interceptors don't have this problem, because they *encapsulate* their delegate.

The direct model provides no way to inform an SO that a call has gone to its delegate (later models will remedy this fault). The only way to prevent the SO's state from becoming stale is to disallow some SO methods (we cannot disallow O.delete because of the disallow constraint). We might think to disallow SO.getColor(x), since it is the method that revealed the problem in our example. But this does not fully solve the problem, because this doesn't prevent the SO state from becoming stale. After the TF runs, the SO is replaced by the current object and its stale state will be revealed, since at this point getColor is allowed. Therefore, we must instead disallow the method that allows the inconsistency to happen in the first place: SO.insertColor. If we disallow this method, then *SO.colorMap* will always be empty, and the invariant cannot be violated. From the callers' point of view, elements of the ColorSet will always have the color blue (until the TF runs), and no sequence of calls to the delegate or SO can cause an inconsistency. In fact, we could allow calls to SO.insertColor(x, blue); by the same reasoning, no inconsistency is possible.

The second problem with the direct model is that the effects of an SO method may be *unimplementable* on the delegate. This may be because the method is a mutator whose shadow is *inexpressible* (Section 3.6), i.e., there's no way to express the effects of the shadow as a sequence of the delegate's methods. In this case, the SO method is often disallowed anyway (for the reasons discussed in Section 3.6). However, certain observers and even some expressible mutators may be impractical to implement because the delegate's type is *behaviorally incomplete* [65]. Informally, this means the delegate does not provide "full access" to its datatype. For example, consider an SO that implements IntSetWithSize (IntSet with a size method) by delegating to an object that implements IntSet. The delegate provides only one observer, contains. The only way for the SO to determine how many elements are in the IntSet is by calling contains on every possible integer, of which there may be infinitely many (if these are arbitrary-precision integers). Since this is impossible (or at least impractical), the SO must disallow size.

4.4 Hybrid Model

When upgrades are compatible, the interceptor model is easy to reason about and implement. But it is impractical for incompatible upgrades due to modularity problems. The direct model is practical for all kinds of upgrades, but it is much less powerful than the interceptor model. For example, an SO in the direct model cannot even simulate ColorSet on IntSet well, even though ColorSet is a subtype of IntSet!

Neither model is ideal in all cases, but we can do somewhat better using a hybrid approach. The idea is to use interceptors when possible, and *non-interceptors* otherwise (where non-interceptors are SOs that receive calls for only their own type, as in the direct model). This does not mean simply using interceptors for compatible upgrades and non-interceptors for incompatible upgrades, because we also want to use interceptors for incompatible upgrades when this is practical, i.e., when the interceptor just has to implement the old and new types and no others.

After a node installs an incompatible upgrade, it supports the old type using a past SO that runs as a non-interceptor. We cannot allow both past SOs and future SOs to run as interceptors, because only one object can intercept calls for the current type. We choose to provide this extra power for future SOs because we expect compatible upgrades to be the common case, so we want to be able to simulate subtypes using interceptors. We will reexamine this decision in later sections.

4.4.1 Rules

The hybrid model introduces a complication: once there is an incompatible future SO, we can't add more interceptors to the chain. This is because an interceptor that follows an incompatible SO is only prepared to handle the incompatible SO's type, not the one that precedes it (if the interceptor had to handle all earlier types, we would have the same modularity problem that caused us to consider non-interceptors in the first place).

It is unsafe to run an interceptor as a non-interceptor, because this violates the assumption made by the implementor that the interceptor receives all calls. Consider this scenario: the current object implements FlavorSet, the first (incompatible) future SO implements IntSet (and intercepts the FlavorSet calls), and the second (compatible) future SO implements ColorSet. The ColorSet SO expects to intercept all calls, but calls for FlavorSet must go to the IntSet SO, because the ColorSet SO cannot handle them. This means the ColorSet SO cannot keep its colors in sync with the underlying set of integers, and this may violate the expectations of clients.

We could handle this problem by delaying the installation of interceptors (compatible or incompatible) until the latest object is the current object or is an interceptor for a compatible upgrade. But delaying the installation of SOs is a problem, because we have assumed that nodes can always install the future SOs for new upgrades immediately—this property allows us to schedule node upgrades however we want.

We could address this problem by requiring that each upgrade provide two future SOs: one that runs as an interceptor (when possible), and another that runs as a non-interceptor (and disallows calls when necessary). But we don't want the upgrader to have to provide two implementations. Since the only difference between the two implementations is that some calls may be disallowed in the non-interceptor, we allow the upgrader to provide a single implementation that indicates this by marking those methods as *interceptOnly*. The node automatically causes calls to those methods to fail when the SO is a non-interceptor. The SO can also decide which calls to disallow at runtime, using a flag provided by the node that indicates whether the SO is an interceptor.

To summarize: each future SO runs either as an interceptor or not. The system has an invariant: if any future SO is a non-interceptor, then so are all more recent SOs.

When the system installs a future SO, it installs it as an interceptor if possible, else not. This is possible if either there are no future SOs or if the most recent SO is running as an interceptor and is for a compatible upgrade. When the node upgrades, the system switches future SOs to run as interceptors if possible, moving up the chain from the current object to the most recent future SO. This means once a future SO becomes an interceptor (and starts depending on this fact to manage its state), it will remain an interceptor until an upgrade replaces the SO with the current object.

There are past SOs for incompatible upgrades but not for compatible upgrades. We implement calls from a past SO to its delegate as follows: if the call is intended for the current object and there are future SOs running as interceptors, the call goes to the most recent interceptor. This is safe, because the interceptor implements the type expected by the past SO.

Figure 4-4 depicts how we manage SOs in the hybrid model.

Example: Simulating ColorSet on IntSet

Figure 4-5 presents pseudocode for a future SO that implements ColorSet by delegating to IntSet. The SO keeps a map, `colors`, from integers to colors, with the assumption that an integer without a color in the map has the color blue. The SO simply delegates insert and contains. The SO implements insertColor and getColor according to their specifications; insertColor is labeled *interceptOnly*, because the SO cannot keep the colors in sync with the IntSet unless it is an interceptor. Finally, the SO implements delete appropriately, i.e., so that deleting an element and re-inserting it will restore its color to blue.

4.4.2 Discussion

Most software changes preserve or extend the behavior of a system, rather than removing behavior. Therefore, we expect compatible upgrades to be the common case, i.e., most upgrades will

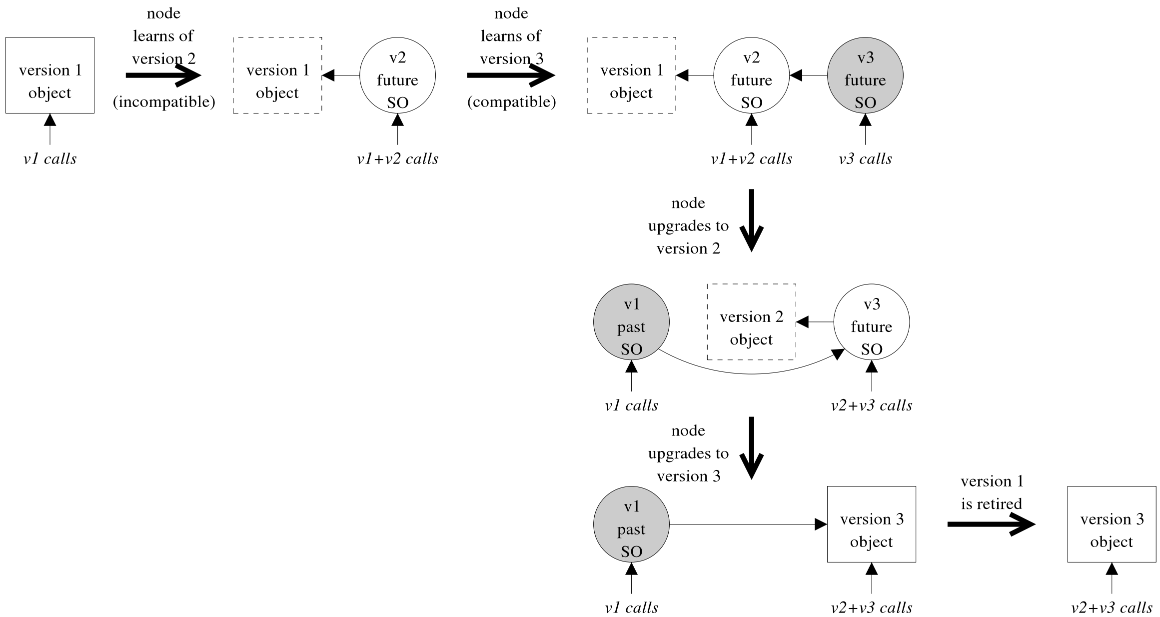


Figure 4-4: *The hybrid model. White SOs are interceptors; grey are non-interceptors. Version 2 is an incompatible upgrade: its future SO is an interceptor that implements versions 1 and 2, and its past SO is a non-interceptor for version 1 only. Version 3 is a compatible upgrade: its future SO is an interceptor when possible, but it runs as a non-interceptor before the node upgrades to 2. After the upgrade, the SO intercepts all calls for versions 2 or 3, including those from the version 1 past SO.*

not change a node's type, and those that do will extend the type in compatible ways. The hybrid model works well in this case, because compatible SOs run as interceptors. However, incompatible upgrades are problem: they require that we use non-interceptors to support later future versions and past versions.

Because past SOs are so weak, clients that upgrade before their servers (and use future SOs) get better service than clients that upgrade after their servers (and use past SOs). But this does not match reality: servers typically upgrade eagerly, while clients upgrade only when necessary. This is because servers are under the direct control of the service provider, while clients are under the control of users.

We could address this problem by reversing the hybrid model (past SOs are interceptors and future SOs are non-interceptors), as depicted in Figure 4-6. This works well when servers upgrade ahead of clients and don't talk to each other. However, this approach has modularity problems, because each upgrade must define a past SO that supports all previous versions of the node (down to the latest retired version). This is because once a past SO is running as an interceptor, we cannot later switch it to run as a non-interceptor (doing SO could invalidate the state of the past SO).

```
class ColorSetSO implements ColorSet:
  ColorSetSO(IntSet next, boolean isInterceptor):
    colors = new Map() // empty map; means all colors are "blue"

  delegate insert(x)

  delegate contains(x)

  interceptOnly void insertColor(x, c):
    if not contains(x):
      insert(x)
      colors.put(x, c)

  color getColor(x, c):
    if not contains(x):
      throw NoSuchElementException
    if x in colors:
      return colors.get(x)
    return "blue"

  void delete(x):
    colors.remove(x)
    next.delete(x)
```

Figure 4-5: *Pseudocode for a* ColorSet *future SO*

Furthermore, this model does not address the problem of upgrading server-to-server systems in which every node is a client of every other node. In these systems, some nodes will always be ahead of others, so both past and future SOs are necessary.

Another approach is to convert an incompatible upgrade into a compatible one by dividing it into two stages. The first stage is a compatible upgrade that replaces the old type with a common subtype of the old and new types. This stage also changes clients to use just the new type. This upgrade can happen gradually, since non-upgraded clients can use upgraded servers directly (i.e., without a past SO).

The second stage is a supertype upgrade that removes support for the old type (i.e., the deprecated methods). This upgrade occurs after the clients have upgraded (in the first stage) and so can use a trivial past SO that disallows all calls.

The two-stage approach lets us avoid biasing our model toward past or future SOs and therefore works for server-to-server systems. However, it relies on the ability to define a common subtype between the two types, which may be impossible if the two types have conflicting history properties, i.e., if calls had to be disallowed for the reasons discussed in Section 3.4.5. If no calls were disallowed for such reasons, then the common subtype is just a "union" of the old and new types: it

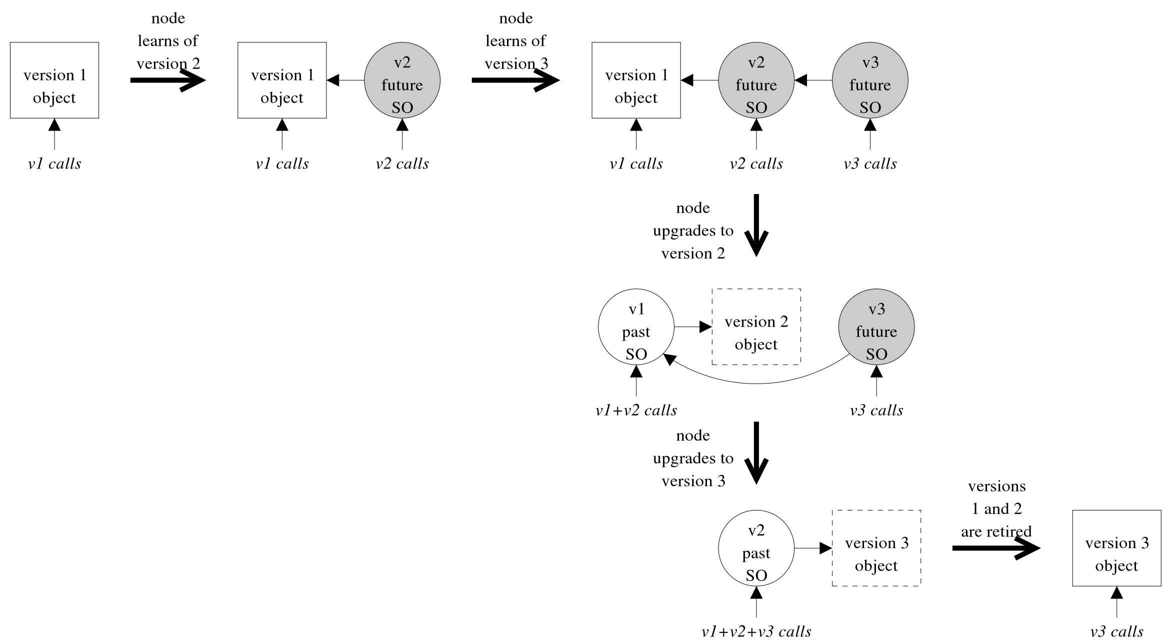


Figure 4-6: *The reverse-hybrid model. Future SOs (grey) are non-interceptors; past SOs (white) are interceptors. Future SOs just implement their own version. Past SOs implement their own and all previous (non-retired) versions.*

has the abstract states and methods of both types (possibly renamed to avoid conflicts), and reflects method calls on both states (according to the shadow methods).

The two-stage approach requires that we delay the second stage until all clients have completed the first one. If we're willing to delay later upgrades, we can use a much simpler model that allows both past and future SOs to run as interceptors (at different times) and yet preserves modular reasoning.

Figure 4-7 depicts the delay-hybrid model. In this model, all SOs run as interceptors (eventually). But as in the hybrid model, we cannot install more future SO interceptors once there is an incompatible SO in the chain, so we run them as non-interceptors when this is the case. Unlike the hybrid model, past SOs run as interceptors, so we cannot run future SOs as interceptors at the same time (because only one object can intercept calls for the current type). Furthermore, we cannot install later upgrades while a past SO exists, because the past SO cannot handle calls for those later versions. Therefore, we must delay later upgrades until the past SO is retired, i.e., until *all* nodes have installed the incompatible upgrade. This means we always have at most one past SO.

The model is practical when the time between upgrades exceeds the time it takes for all the nodes to install an upgrade. This is likely to be the case for server clusters in which the service

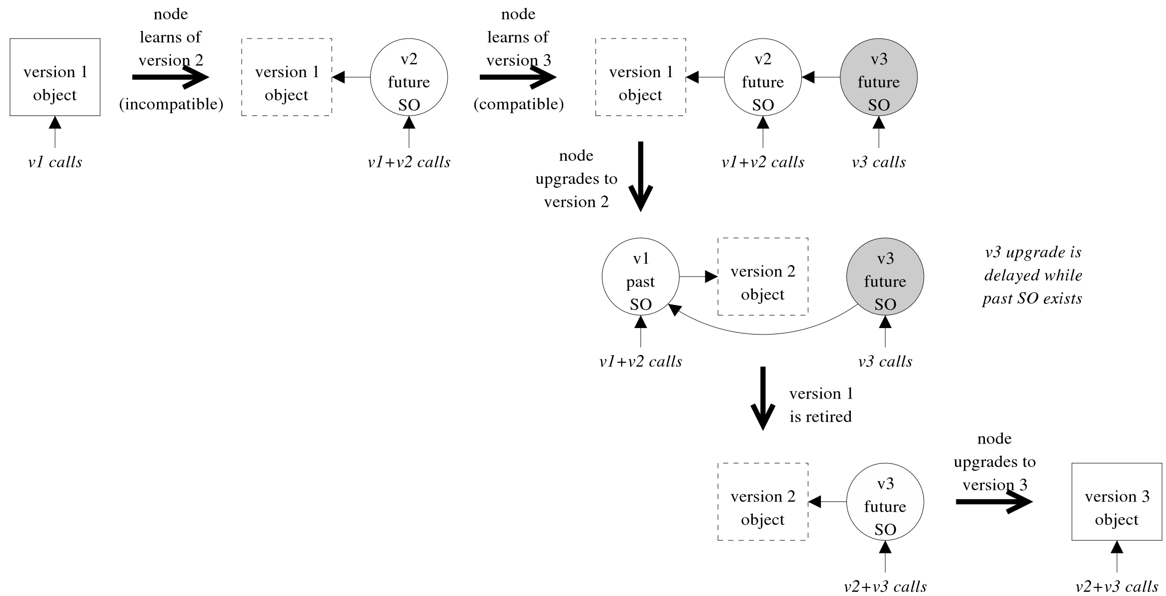


Figure 4-7: *The delay-hybrid model. White SOs are interceptors; grey are non-interceptors. Past SOs are always interceptors, and at most one past SO can exist at a time.*

provider can cause all the nodes to upgrade in the course of a few hours, but this won't work if clients can control when nodes upgrade.

If we want to avoid restricting the upgrade schedule in any way (either with delays or by upgrading clients before servers or vice versa), then we need a more powerful simulation model. We present such a model in the next section.

4.5 Notification Model

In this section, we propose a model that allows for powerful past and future SOs at the cost of requiring cooperation from the current object. The idea is for each object on a node (whether it is an SO or the current object) to notify all the other objects on the node when the object receives a method call. This notification takes the form of a special method call on a *notification interface* implemented by the other objects. The notification for a method *m* has the name *notifyM* and has the same arguments and return value as *m*. The purpose of the notification is to allow the receiver to reflect the effects of the method call in its own state; these effects are exactly as specified by the shadow of that method on the receiver. Thus, notifications are only needed for mutators.

We don't want to require that every object understand notifications from every other object; this would not be modular, and furthermore, we don't know what notifications to expect from future

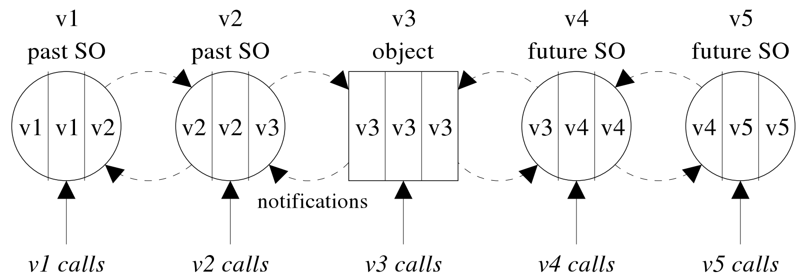


Figure 4-8: *The notification model. Each object has three interfaces: a normal interface (in the middle) that accepts calls from clients (solid arrows), a past notification interface (on the left), and a future notification interface (on the right) that each accept notifications from neighbors (dotted arrows). The version numbers on the interfaces indicate which mutators the interface accepts, e.g., "v2" means that interface accepts calls or notifications for version 2 mutators. Not shown: each SO also can also call observers of its delegate (the next object toward the current version).*

versions when we implement the current object and future SOs. Instead, each object just understands notifications for its own mutators and, in the case of SOs, the mutators of the next object closer to the current version. These are exactly the old and new types the upgrader must already consider when defining an upgrade, so this design has good modularity.

Each object has three interfaces: a *normal* interface, which is called by clients; a *past* interface, which accepts notifications from the object of the next older version, and a *future* interface, which accepts notifications from the object of the next newer version. The mutators for which the past and future interfaces accept notifications depends on the kind of object: the current object only accepts notifications for its own mutators on either interface, i.e., its past and future interfaces accept *notifyM* for each mutator *m* of the current type. A past SO accepts notifications for its own mutators on its past interface and accept notifications for the next newer type's mutators on its future interface. A future SO accepts notifications for its own mutators on its future interface and accept notifications for the next newer type's mutators on its past interface. This arrangement is symmetric, as depicted in Figure 4-8.

When an object receives a call on its normal interface, it modifies its state as needed, notifies the object of the next newer version on that object's past interface (if that object exists), and notifies the object of the next older version on that object's future interface (if that object exists). Those objects in turn modify their state as needed and propagate the notification to the next newer and older objects, respectively. Thus, notifications propagate outward from the object that receives the original call.

But the notifications change as they propagate. For example, suppose the current object implements ColorSet (Figure 3-2, page 41), and the past SO implements IntSet (Figure 3-1, page 33). When the current object receives the call insertColor(x, blue), it simply calls notifyInsertColor(x, blue) on its two neighbors (the nearest past SO and future SO). Let's consider just the past SO for now. It modifies its own state according to the shadow of insertColor on its type. But then it must propagate the notification to the next older object, and that object may not understand notifyInsertColor. That object is prepared for notifications on the past SO's type (IntSet). Therefore, the past SO must translate insertColor(x, blue) into a sequence of mutator calls on its own type, e.g., insert(x), and notify the next older object as though those mutators were called, e.g., notifyInsert(x). Similarly, when the past SO receives a notification on its past interface, e.g., notifyDelete(x), it must translate delete(x) into a sequence of mutator calls on the next newer type (in this case, it's also delete(x)) and notify the next newer object appropriately.

4.5.1 Disallowed Calls

The notification approach requires that shadow methods are *expressible* (Section 3.6) and *implementable* (Section 4.3.1), because objects that receive a notification for an inexpressible or unimplementable method will be unable to translate the notification and propagate it further. The automatic disallowing discussed in Section 3.6 handles this problem for inexpressible methods: it disallows calls to methods whose shadows are inexpressible or disallows calls to those types that cannot reflect the effects of those shadows. However, this does not address the problem for unimplementable methods. The simplest solution is to treat unimplementable methods as inexpressible and disallow them similarly, but this may be overly conservative. Weakening this restriction is future work.

4.5.2 Implementing Notifications

The notification model may sound complicated, but it can be implemented quite simply. We demonstrate how with an example.

Figure 4-9 gives pseudocode for the normal interface of a ColorSet future SO whose delegate is an IntSet. Figure 4-10 gives its notification interfaces.

The SO's observers are implemented by calling methods of its delegate, just as in the previous models. For example, ColorSet.contains delegates to IntSet.contains.

```
class ColorSetFutureSO implements ColorSet:
  ColorSetFutureSO(IntSet next, NotifyIntSet past, NotifyColorSet future):
    colors = new Map() // empty map; means all colors are "blue"

  void insert(x):
    past.notifyInsert(x)
    future.notifyInsert(x)

  boolean contains(x):
    return next.contains(x)

  void insertColor(x, c):
    if not contains(x):
      colors.put(x, c)
    past.notifyInsert(x)
    future.notifyInsertColor(x,c)

  color getColor(x, c):
    if not contains(x):
      throw NoSuchElementException
    if x in colors:
      return colors.get(x)
    return "blue"

  void delete(x):
    colors.remove(x)
    past.notifyDelete(x)
    future.notifyDelete(x)

  class Past implements NotifyIntSet:
    // see next figure

  class Future implements NotifyColorSet:
    // see next figure
```

Figure 4-9: *Pseudocode for the normal interface of a* ColorSet *future SO*

```
class Past implements NotifyIntSet:
  void notifyInsert(x):
    future.notifyInsert(x)

  void notifyDelete(x):
    colors.remove(x)
    future.notifyDelete(x)

class Future implements NotifyColorSet:
  void notifyInsert(x):
    past.notifyInsert(x)

  void notifyInsertColor(x, c):
    if not contains(x):
      colors.put(x, c)
    past.notifyInsert(x)

  void notifyDelete(x):
    colors.remove(x)
    past.notifyDelete(x)
```

Figure 4-10: *Pseudocode for the notification interfaces of a* ColorSet *future SO*

The SO's mutators are implemented differently from the previous models: the SO no longer calls mutators of its delegate (in fact, the `next` reference does not allow mutations). Instead, all mutations are propagated via notifications.

The SO is constructed with references to the notification interfaces of its two neighbors: the future interface of its past neighbor and the past interface of its future neighbor. (If a neighbor is missing, a stub interface that ignores notifications is substituted.) The SO implements its `Past` and `Future` notification interfaces as non-static inner classes, meaning each SO instance has an instance of each inner class, and those instances can access the SO's private fields directly. Thus, the SO's past neighbor has a reference to the `Past` instance, and the SO's future neighbor has a reference to the `Future` instance.

The implementations of the three interfaces of the SO have much in common. For example, `ColorSetSO.delete`, `Past.notifyDelete`, and `Future.notifyDelete` differ only in how they propagate notifications. What's really going on here is that each SO mutator can be thought of as having three parts: a local implementation, a future notification (which is trivial), and a past notification (which requires translation to the past neighbor's type). If the SO implementor provides these three parts separately, a compiler can easily generate the three interfaces of the SO. Implementing such a tool is straightforward; we leave it for future work.

```
class IntSetWrapper implements IntSet:
  IntSetWrapper(IntSet actual, NotifyIntSet past, NotifyIntSet future):
    // nothing in constructor

  void insert(x):
    actual.insert(x)
    past.notifyInsert(x)
    future.notifyInsert(x)

  boolean contains(x):
    return actual.contains(x)

  void delete(x):
    actual.delete(x)
    past.notifyDelete(x)
    future.notifyDelete(x)

  class Past implements NotifyIntSet:
    void insert(x):
      actual.insert(x)
      future.notifyInsert(x)

    void delete(x):
      actual.delete(x)
      future.notifyDelete(x)

  class Future implements NotifyIntSet:
    void insert(x):
      actual.insert(x)
      past.notifyInsert(x)

    void delete(x):
      actual.delete(x)
      past.notifyDelete(x)
```

Figure 4-11: *Pseudocode for a notification wrapper for* IntSet

Figure 4-11 presents pseudocode for a wrapper that handles the notifications for the current object, IntSet. The wrapper implements the IntSet methods by delegating to the `actual` object, and it propagates notifications to the neighboring past SO and future SO on their future and past interfaces, respectively. The innermost past SO is initialized with a reference to the current object's `Past` interface (as its `future`), and the future SO is initialized with a reference to the current object's `Future` interface (as its `past`).

The benefit of using a wrapper is that we can support the notification model without actually modifying the current object. Furthermore, a compiler could easily generate this wrapper given the IntSet interface.

4.5.3 Discussion

The benefit of the notification model is that past SOs and future SOs are equally powerful. This affords us greater flexibility in scheduling upgrades, since there is no inherent bias toward upgrading clients before servers or vice versa. However, the Achilles heel of this model is concurrency control: without synchronization, notifications traveling up and down the chain may be interleaved arbitrarily. We discuss this further in the next section.

4.6 Concurrency Control

So far, we have assumed calls are applied to a node in some serial order, and each call is allowed to complete before the next one begins. But in reality, a node may process many calls concurrently. We want to allow such concurrency, because enforcing serialization can reduce performance and, if calls block, can cause deadlock.

However, we don't want to dictate the exact method of concurrency control, because no solution we provide can satisfy all applications. We considered providing various primitives, such as critical sections or locks, but all of them have problems: they introduce the risk of deadlock, and they are too coarse-grained for some applications.

Our solution is to let the objects on a node—the current object and simulation objects—implement synchronization themselves. These objects can be running many calls in parallel, and they synchronize them somehow. This might mean complete serialization, i.e., a call completes executing on all objects before the next call is dispatched to any object. But this is probably too strict for many applications, and we want to give upgraders the flexibility to choose their own synchronization policy.

Implementing synchronization is straightforward using interceptors, because interceptors handle all calls that affect their state and so can control the order in which they are applied. This makes the hybrid model and its variants attractive for applications that require synchronization.

Non-interceptors (such as past SOs in the hybrid model, and all the SOs in the direct and notification models) cannot control how calls are applied to their delegates, so a non-interceptor cannot execute an atomic sequence of calls on its delegate unless the delegate provides some concurrency control mechanism.

For example, suppose the current object implements a queue with methods enq and deq, and the future SO implements a queue with an additional method, deq2, that dequeues two consecutive items. If the SO is an interceptor, it can implement deq2 simply by calling deq twice on the delegate and ensuring no other deq calls are in progress. But a non-interceptor cannot do this, because another client could call deq in between the non-interceptor's deq calls.

One way to address this is for the delegate to provide some form of application-level concurrency control. For example, the delegate may provide a lockdeq method that locks the queue on behalf of the caller for any number of deq calls, but allows enq calls from other clients to proceed. The non-interceptor can use lockdeq to implement deq2 correctly, e.g.:

```
Pair deq2():
  next.lockdeq()
  e1 = next.deq()
  e2 = next.deq()
  next.releasedeq()
  return new Pair(e1, e2)
```

But if the delegate does not provide appropriate concurrency control methods, then our only choice is to disallow `deq2`.

The problem of synchronization is worse in the notification model. Notifications propagate in all directions and may be interleaved, and there is no analogue of an interceptor that can control the order in which calls execute on different objects.

We might hope to address the problem with application-level concurrency control, but this is complicated. For example, an object that receives deq2 must notify both its neighbors of this call. This is straightforward if the neighbors support `notifyDeq2`. But a neighbor might only understand deq, in which case an auxiliary method like lockdeq is needed. As the notification propagates, the sequence of calls needed to propagate its effects can grow longer, and more synchronization will be needed to keep its effects atomic. This is unlikely to work beyond a few objects, which means calls will need to be disallowed. This suggests that the only practical concurrency control solution for the notification model is complete synchronization (one call executes at a time), but this is too inefficient for real systems.

4.6.1 Failures

What happens when a call fails in the middle of a sequence of calls made by an SO to its delegate? Depending on the specification of the method the SO was trying to simulate, it may have to roll back the effects of the calls to the delegate that succeeded. For example, suppose an SO implements deq2 by calling deq twice on its delegate, and the first call succeeds but the second one fails. Then, the entire deq2 operation should fail, which means the deq that succeeded must be undone.

Interceptors can implement these semantics by keeping state to record the results of partial operations. For example, an interceptor can implement a failure-tolerant deq2 with a one-element local buffer:

```
Pair deq2():
  if local == null:
    local = next.deq()
  p = new Pair(local, next.deq())
  local = null
  return p
```

However, non-interceptors cannot keep such state, because calls may go directly to the delegate and observe partially-mutated state. Non-interceptors may still be able to implement failure-tolerant sequences if the delegate provides methods to execute atomic sequences of calls, i.e., transactions. But if the delegate does not provide a way to implement atomic sequences, non-interceptors must disallow methods that require them, like deq2.

4.7 Discussion

In this chapter, we have discussed several models for managing simulation objects. No one model is perfect, so different models are suitable for different deployment scenarios and upgrade patterns.

Table 4.1 compares how well the various models support several desirable characteristics:

Future simulation Nodes can simulate future specifications well (i.e., few calls are disallowed).

Past simulation Nodes can simulate past specifications well (i.e., few calls are disallowed).

Simulation Model	Future Simulation	Past Simulation	Concurrency Control	Modular Reasoning	No Delays
Interceptor	•	·	•	·	•
Direct	·	·	·	•	•
Hybrid	◦	·	◦	•	•
Reverse-hybrid	·	•	◦	·	•
Delay-hybrid	◦	•	◦	•	◦
Notification	•	•	·	•	•

Table 4.1: *How well the models for using simulation objects support desirable characteristics (listed in the top row). Key: (•) the model supports the characteristic well; (◦) the model sometimes supports the characteristic well; (·) the model supports the characteristic poorly or not at all.*

Concurrency control Nodes can control the order in which calls for different versions are applied.

Modular reasoning Defining an upgrade only requires understanding the old and new versions.

No delays Nodes can upgrade as soon as their SF signals.

The two most powerful models are the delay-hybrid and notification models. Both are modular and provide good support for simulation. None of the other models support both past and future simulation well, so certain scenarios will cause difficulties, e.g., incompatible upgrades in server-to-server systems. In such cases, the system may experience service degradation during upgrades, because some calls will be disallowed.

Delay-hybrid provides adequate concurrency control for SOs at the expense of sometimes delaying upgrades. It is a model to use if each upgrade tends to complete before the next one begins. The notification model offers the best flexibility and power, but it is not practical for systems in which nodes need to execute multiple calls concurrently. If delays are unacceptable and concurrency is required, then the hybrid model is a good choice.

4.7.1 Reasons to Disallow

In Chapter 3 and throughout this chapter, we have discussed several reasons why calls to simulation objects may need to be disallowed. We summarize these reasons here.

The first few reasons to disallow are independent of the simulation model used; they depend solely on the upgrade specification:

History Violations (Section 3.4.5; example: upgrading IntSet to GrowSet) Sometimes it is not possible to reflect a call on both types without violating the history properties of one of the types. In this case, calls that would violate the properties or would reveal the violation must be disallowed.

Inexpressibility (Section 3.6; example: upgrading IntSet to GrowSet) When the shadow of a method is inexpressible, either that method must be disallowed or calls to objects that are unable to reflect the effects of that method must be disallowed.

Disallowing due to history properties is only a concern when an upgrade specifies (in terms of its invariant and shadow methods) that the effects of a call *must* violate the history properties of one of the types. This reason to disallow can be avoided by weakening the invariant between the states of the two types.

The next reason to disallow is not a requirement, but a choice:

Implementation Complexity (Section 3.5; example: upgrading PermServer to AclServer) Sometimes providing the full behavior of a type is more trouble than it's worth. An upgrader may choose to disallow certain calls of a type to simplify the implementation of simulation objects for that type, especially when disallowing those calls allows the SO to be stateless. This simplifies not only the SO implementation but also that of the transform function, since it means the TF operates only on the state of the current object.

The remaining reasons to disallow apply only to non-interceptors:

Interleaving (Section 4.3.1; example: simulating ColorSet on IntSet) When a call bypasses an SO and goes directly to its delegate, the SO may be unable to reflect the effects of the call. In this case, the SO must disallow calls that could cause its state to become inconsistent with that of the delegate.

Unimplementablility (Section 4.3.1; example: simulating IntSetWithSize on IntSet) When an SO receives a call, it must cause its delegate to reflect the effects of that call by calling that delegate's methods. If the method is inexpressible or if the delegate's type is behaviorally incomplete, the SO cannot implement the call correctly and must disallow it. An analogous problem occurs in the notification model when an SO is unable to translate a notification it receives into notifications that its neighbors understand (Section 4.5.1).

Concurrency (Section 4.6; example: simulating deq2 using deq) When an SO must make more than one call to its delegate to reflect the effects of a call to the SO, it is possible that the delegate may receive other calls in between those made by the SO. If this would cause the delegate to reflect the SO's call incorrectly, the SO must disallow the call it received.

Atomicity (Section 4.6.1; example: simulating deq2 using deq) When an SO must make more than one call to its delegate to reflect the effects of a call to the SO, it is possible that some of those calls may fail. This may require that the SO roll back the effects of the calls that succeeded. If it is unable to do so, the SO must disallow the call it received.

Deciding which calls to disallow can be difficult. Thankfully, there are several ways to simplify the process. Disallowing due to interleaving is only needed when state of the SO can become stale; this is not a problem for stateless SOs. Disallowing due to concurrency issues occurs only when the delegate's type provides insufficient means to do concurrency control, but many types provide synchronization primitives in the form of application-level locking and transactions. Finally, most of these reasons to disallow can be avoided altogether using interceptors, and many of the models discussed above allow SOs to run as interceptors most of the time.

Chapter 5

Transform Functions

After a node has learned of an upgrade, but before it starts running the new class, it implements its current type with an instance of its current class and (depending on the simulation model) implements the new type with a future SO. After the upgrade, the node implements the new type with an instance of the new class and (depending on the simulation model) implements the old type with a past SO. The job of the transform function (TF) is to reorganize the persistent state of a node from the representation required by the old instance and future SO to that required by the new instance and past SO, as depicted in Figure 5-1.

A transform function runs after a node has been shut down, so it operates only on a node's persistent state; the node's volatile state is discarded. Nodes can recover from just their persistent state, so this is safe. This design means nodes can manage their volatile state however they like; in particular, node software is not restricted to particular languages, as in dynamic updating sys-

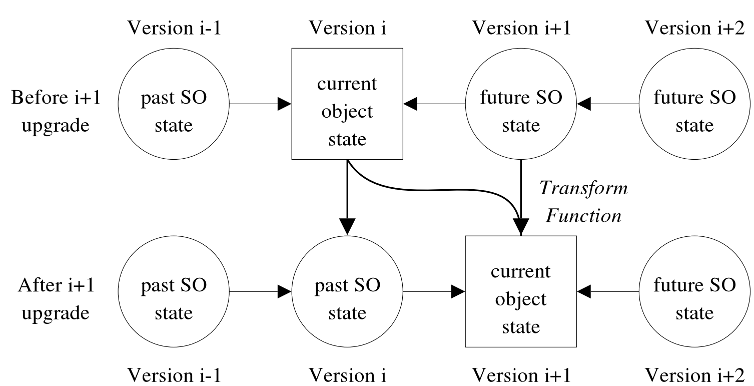


Figure 5-1: *Transform function for a node upgrade from version i to i+1. The TF implements the identity mapping, so clients of both versions observe the same state after the TF runs as before.*

tems [46,48,54,57,58,75]. Furthermore, node software need not provide explicit methods to export its volatile state [29, 48, 59]. Thus, our approach allows for simpler implementations of both the node software and the TF.

The transform function implements the identity mapping for the old and new abstract states: the abstract state of the past SO (after the TF runs and the node recovers) is the same as that of the old instance before the TF ran, and the abstract state of the new instance (after the TF runs and the node recovers) is the same as that of the future SO before the TF ran. Thus, clients of the node do not notice that the node has upgraded, except that clients of the new type may see improved performance and fewer rejected calls, and clients of the old type may see decreased performance and more rejected calls.

A transform function involves only local code; it may not call methods on other objects. We cannot allow the TF to rely on other nodes being able to handle the calls the TF might make, because we can make no guarantees about when one node upgrades relative to another. If the node being upgraded needs to obtain state from another node (e.g., in a replicated system), it should transfer this state after it has completed the upgrade, not during the TF. This helps avoid deadlocks that may occur because nodes upgrading simultaneously attempt to obtain state from each other. This also makes transform functions simpler to implement and reason about.

Section 5.1 presents our model for how nodes use their persistent state, how they recover, and how transform functions fit into this design. A transform function must satisfy certain requirements so that it works properly in an asynchronous environment with failures; we discuss these in Section 5.2.

In Section 10.3.1, we discuss a related piece of future work: *incremental transform functions*. Incremental TFs transform parts of a node's state as they are accessed, rather than transforming the entire state at once. Incremental TFs can reduce node downtime, but they add complexity to the system and are not always feasible.

5.1 Base Model

We assume objects have access to persistent storage, which they use either directly, e.g., by reads and writes to disk, or through some storage layer such as a file system or database. Objects are responsible for writing to persistent storage as they carry out methods in order to ensure whatever stability is needed to satisfy their specification. They may also run code in between method calls to reorganize this storage.

We also assume that objects that use persistent storage provide a *recover* method that is called by the node (not by the upgrade system) when it has recovered from a failure. This method reads the persistent state as needed to initialize the volatile state of the object so that it can carry out method calls. The recover method also "cleans up" the persistent state to remove any partially-executed operations. For example, the recover method of a database restores the database state on disk from an on-disk transaction log, rolls back any partially-executed transactions, and initializes the database's in-memory tables [80].

Our idea is to insert the transform function in the middle of the recovery process without interfering with normal operation. When a node is ready to install an upgrade, the upgrade system sets a persistent *transform-flag* and causes the node to restart. After the restart, the node checks whether the *transform-flag* is set and, if so, calls the transform function for the pending upgrade rather than the recover method. The TF runs the old version's recover method to bring the node to a pristine, "newly-recovered" state. If there is a future SO that has persistent state, the TF also runs its recover method (as described in Section 5.1.1). Then, the TF reorganizes the persistent state as needed. When the TF finishes, the upgrade system clears the *transform-flag* and runs the recover method for the new version. In the common case when the TF does nothing, we need not run both the old and new recover methods; we can just run the new one.

We have a persistent *transform-flag* so that recovery survives node failures. If a node fails while a transform is in progress, the transform may be partially complete when the node recovers. In this case, the *transform-flag* will still be set, so the node can simply restart the transform (hence our requirement that TFs are restartable, as discussed in Section 5.2.3). This is a conservative approach, since we may restart the TF unnecessarily if the node fails after the TF completes but before we clear the *transform-flag*. Such an occurrence should be extremely rare, and an intelligent TF can detect that the transform is already complete and avoid redundant work.

5.1.1 Recovering SOs

This idea of recovery needs to be adjusted somewhat to take into account the fact that we have simulation objects in addition to the current object, and each of these objects may have its own persistent state.

Each object in the chain of objects must be given a chance to recover. If an SO doesn't have persistent state of its own, its recover method does nothing. If the SO has persistent state, it must manage a persistent store and must provide a recover method, and the node must call the SO's recover method on restart.

Order

Objects must be recovered in the proper order. First the current object is recovered. Then the future SOs are recovered in version number order, starting from the current object and moving up the chain of future SOs. The node provides the future SO with a reference to its delegate and (depending on the simulation model) a flag indicating whether the SO is running as an interceptor. This order enables a future SO to call methods of its delegate as needed to do recovery.

After all future SOs have been recovered, past SOs are recovered in reverse version-number order, i.e., most recent first. We recover future SOs first so that calls made by past SOs to the current object can be redirected to future SOs running as interceptors. If past SOs were the interceptors, we would recover them first.

Since we recover the current object first, we do not need to change its recovery code. However, the upgrade system must recover the SOs before it allows the current object to receive calls from any outside clients.

Initialization

When a future SO is first installed, it must create its persistent state, if it has any. The initial value of a future SO's abstract state is the result of applying the mapping function to the abstract state of the SO's delegate at the moment the SO is installed.

We do not want the installation of a future SO to stall the node, so we do not provide any synchronization for the initialization of the SO. This means the node continues to run while the future SO initializes its state, and the initial value of its state is a moving target. This limits the kind of initialization that can be done.

The future SO may need to do some disk activity to prepare its persistent state. For example, a ColorSet future SO may keep a persistent map from integers to colors, and it needs to initialize this map to the default colors. Since the future SO does not get a chance to examine the state of its delegate when it is installed, it just initializes the disk blocks for its state to null. The SO initializes its state with actual values lazily, i.e., as it observes state of its delegate in the course of normal processing. For example, the ColorSet SO can wait to initialize the color for an integer until that integer is accessed.

Lazy initialization is effective when the future SO can construct its state piecemeal from that of its delegate, but not when the state must be constructed all at once. And while lazy initialization can reduce the node's recovery time, it introduces runtime overhead because the SO may need to access the disk on each method call.

If a future SO cannot construct its state lazily, it should wait to initialize its state until it can run as an interceptor (until then, the SO must disallow calls that access persistent state). Once the SO is running as an interceptor, it can serialize its initialization among the calls to the node. In particular, the SO could block mutators to its delegate (but allow observers) while it initializes its state.

A past SO might also have persistent state, but this is not initialized while the node is running. Instead, the TF creates the persistent state for the past SO in addition to creating the persistent state for the new object. For example, if the node upgrades from ColorSet to FlavorSet, the TF creates an integer-color map for the past SO in addition to creating the integer-flavor map for the new object. The TF could do this just by leaving the old persistent state as-is and creating a completely new persistent state for the new object, i.e., the old persistent state becomes the persistent state of the past SO. Alternately, the TF could create a new persistent state for the past SO.

5.2 Requirements

We have three requirements for transform functions. First, we require that upgrades are transparent to clients. Second, a TF must be able to run at any time, i.e., not just when the node is in a particular state. Third, a TF must work across node restarts, since the node may fail while the TF is in progress.

5.2.1 Transparency

We require that clients do not notice when a node upgrades. This is relative to the specifications that the clients use, and it includes whatever the specifications say about persistence. For example, if some of the object's state is not persistent, that part will be reset when the node upgrades.

After the upgrade, a client that was using the current object now uses the past SO. What the client sees must be consistent with its previous uses of the current object. For example, if node upgrades from GrowSet to IntSet, the client will not observe that any objects disappear from the set (as discussed in Section 3.4.5). This is either because the past SO disallows all observers (the invariant is $O_{old} = O_{new}$) or because the past SO has persistent state that records the elements of the set (the invariant is $O_{old} \supseteq O_{new}$). In the latter case, the TF must initialize the state of the past SO to contain the recorded elements.

Similarly, a client that was using the future SO now uses the current object. What the client sees must be consistent with its previous uses of the future SO. For example, if node upgrades from IntSet to GrowSet (the reverse of the above), the client again will not observe that any objects disappear from the set. This is either because the future SO disallowed all observers (the invariant is $O_{old} = O_{new}$) or because the future SO had persistent state that recorded the elements of the set (the invariant is $O_{old} \subseteq O_{new}$). In the latter case, the TF must preserve these elements in the state of the current object.

Clients may observe changes outside the specification. In particular, clients that start using the current object may see better performance and fewer rejected calls, and clients that start using the past SO may see degraded performance and more rejected calls.

5.2.2 Runs At Any Time

A transform function must be prepared to run at any point in a node's computation, although the TF can assume that the node is in the "newly-recovered" state when it runs. We might hope to control when the TF runs using the scheduling function, but this might cause the node to wait forever for the node to reach a particular state. Instead, we require that the TF work regardless of when it is invoked; this gives us the flexibility to use scheduling functions that time out or respond to an external signal.

5.2.3 Restartability

A transform function must be *restartable* and *idempotent*, i.e., it must work correctly when terminated and restarted at arbitrary times and when run multiple times [67]. We require restartability because the node may fail while the TF is in progress.

Restartability is easy to implement if the node has enough resources to store the old and new persistent states simultaneously: the TF treats the old state as read-only and writes the new state to new storage. If the TF is restarted, it simply starts over. The upgrade system uses a persistent *transform-flag* (Section 5.1) to avoid restarting the TF unnecessarily.

If the TF must overwrite the old state with the new state, then it must keep some auxiliary information to know how to pick up where it left off, e.g., a log of what parts of the transform have been completed. When the TF restarts, it must be able to produce the remainder of the new state from just the non-overwritten parts of the old state. This is straightforward when the state is composed of records or files that can be modified in-place but is more difficult when the entire storage layer must be reorganized. Existing work in recoverable file systems and databases can be applied here [80, 91].

Chapter 6

Scheduling Functions

Scheduling functions allow us to control how an upgrade progresses through a system. Previous work on automatic upgrades tended to focus on specific systems, so they chose upgrade schedules appropriate for their particular system design [33, 96, 102]. Our design allows upgraders to define different schedules for different class upgrades, thus allowing the upgrader to consider additional factors—like the urgency of the upgrade and how well nodes can interoperate across versions—in defining the upgrade schedule. We begin this chapter with several examples of the kinds of schedules that can be implemented using scheduling functions.

Scheduling functions often require information about the system: Which nodes have upgraded? Which nodes belong to the same replica group? What is the current time? A contribution of our work is an investigation of what kinds of information SFs need and an architecture to provide that information to SFs. In particular, we introduce a central upgrade database that stores the upgrade status of every node in the system and per-node databases that keep track of which nodes are communicating most frequently. We also present a design that allows a scheduling function to access internal state of the node's object without a priori knowledge of which parts of the state the SF will need.

We conclude this chapter with a set of guidelines for designing good scheduling functions.

6.1 Examples

In "Lessons from Giant-Scale Services" [33], Brewer describes various techniques for handling load, failures, and online evolution (upgrades) in giant-scale services. Giant-scale services are large distributed systems that are typically composed of several geographically-distinct data centers, each containing hundreds or thousands of machines. The focus of the article is how different techniques result in different tradeoffs of various service metrics.

Brewer describes three strategies for upgrading giant-scale services. We can express each of these strategies as scheduling functions in our methodology:

Fast reboot "quickly reboots all nodes into the new version;" it is "straightforward, but it guarantees some downtime" [33]. This is not simply an eager scheduling function, since the goal of fast reboot is to cause nodes to upgrade at the same time (not just as soon as they hear about the upgrade). Instead, we can express fast reboot as a scheduling function that signals at a particular time or one that signals in response to a message broadcast by the upgrader.

Rolling upgrade "upgrades nodes one at a time in a 'wave' that rolls across the cluster" [16,33,102]. We can express this approach as a scheduling function that signals if its node has the lowest IP address among the set of non-upgraded nodes. Brewer mentions that "rolling upgrades are easily the most popular" of his three techniques, but "one disadvantage with the rolling upgrade is that the old and new versions must be compatible because they will coexist." In our system, simulation objects allow the upgrader to use rolling upgrades not only for same type upgrades (which is what Brewer means by "compatible" here), but also for subtype and incompatible upgrades.

Big flip "updates the cluster one half at a time by taking down and upgrading half the nodes at once. During the 'flip,' we atomically switch all traffic to the upgraded half using a layer-4 switch ... As with fast reboot, only one version runs at a time" [33]. We can express this approach as a scheduling function that signals at a particular time, depending on whether the node is in the first half or the second half of the cluster. We would still require a layer-4 switch to redirect traffic if we want to avoid breaking connections when nodes upgrade, and clients must be prepared for the loss of volatile state due to node restarts. Both this approach and fast reboot allow the system to avoid cross-version communication, so these are useful when the upgrade does not admit good simulation.

Scheduling functions allow for many other upgrade schedules:

Upgrade eagerly. The SF signals immediately, so nodes upgrade as soon as they download the necessary files (rather than all at once, as in a fast reboot). This schedule is useful to fix a critical bug, but it may disrupt service severely. The Gnucleus [7] file sharing service uses this strategy, since they want to avoid cross-version communication and can afford to disrupt service.

Upgrade gradually. The SF decides whether to signal by periodically flipping a coin. This schedule can avoid causing too many simultaneous node failures and recoveries, and so can limit how many nodes initiate state transfer at once, e.g., in a replicated system. By adjusting the bias of the coin and the period between flips, we can place probabilistic bounds on how many nodes upgrade at once and how long the upgrade will take. We demonstrate the use of this SF in a real Internet deployment in Section 8.2.3.

Upgrade after my servers upgrade. The SF signals once its node's servers have upgraded. This schedule prevents a client node from calling methods that its servers do not yet fully support. This can be used if implementing a future SO for the server is difficult.

Upgrade all nodes of class C1 before any nodes of class C2. The SF for class C2 queries a central database to determine whether all nodes of class C1 have upgraded. This is like the previous example, but it enforces a partial order over the upgrades of all nodes in the system.

Upgrade only nodes 1, 2, and 5 until given the "all clear". This schedule lets the upgrader test an upgrade on a few nodes [96]. The SF signals if its node is one of the allowed ones; otherwise it periodically queries a central node for the "all clear" signal.

Upgrade when the node is lightly loaded. The SF checks the local time, CPU, and/or network load to determine when to signal.

Upgrade without creating blind spots in the geographic layout of the system (e.g., in a sensor network). The SF checks its local position via GPS and queries its neighbors to determine whether it can upgrade without blinding the network. As in a rolling upgrade, nodes order their upgrades using some total order on nodes. Some parts of the network may lack redundant coverage, and nodes in those areas may need to upgrade even when doing so would create a blind spot.

6.2 Inputs

Scheduling functions may require several different pieces of information to decide when to signal. First, the SF may need basic information about the physical node on which it runs. Next, the SF may need to know information about the state of its node's object. Finally, the SF may need information about other nodes in the system and, in particular, the nodes with which its node communicates. In this section, we present our architecture for providing scheduling functions with the information they need.

6.2.1 Node State

A scheduling function may consider the state of the physical node on which it resides, such as its physical location, CPU load, network load, and local time. SFs access these resources via standard operating system interfaces. In addition to the above, the operating system provides access to pseudo-random number generators and periodic timers.

6.2.2 Object State

A scheduling function may consider the state of the current object. In general, we cannot predict what parts of an object's state an SF might need. Instead, we want to provide SFs with read-only access to *all* of a node's state via privileged observers. Restricting SFs to read-only access prevents them from violating the node's specification by mutating its state.

Unfortunately, we cannot rely on the object to know what state the SF will need to access, because the object was implemented before the SF was defined. But in many cases we can predict these requirements based on how the system is designed. For example, we may expect that upgrades for a replicated system will occur round-robin, so it makes sense for the object to provide an observer that returns identifiers for its fellow replicas. Also, we may want to avoid disrupting client sessions when a node upgrades, so it makes sense for the object to provide a way to determine the number of active sessions.

If the object does not provide the necessary observers, the SF could access the persistent state of the node directly, e.g., via the file system. This is dangerous, however, since the object may mutate its state while the SF reads it, so the SF may read inconsistent values.

The SF might also be able to get the information it needs from the future SO. For example, the SO could provide an extra observer that returns the number of open connections. But this approach only works if the SO is running as an interceptor, and it cannot provide information on the internal state of the current object.

We propose a solution that allows an SF to observe arbitrary parts of the object's state without a priori knowledge of which parts of the state the SF will need. This is just a design; implementing it is future work.

Our solution is to generate a privileged *meta-observer* for each object automatically. The meta-observer is a method that accepts a callback as an argument. The meta-observer calls the callback, passing as arguments read-only references to the fields (public and private) of its object. The callback in turn returns to the SF the values of the fields in which the SF is interested. Thus, different SFs can use different callbacks to obtain the information they need, and the meta-observer can be generated without knowing which parts are needed.

Alternatively, we could have generated an observer that simply returns *all* the object's fields, but this would be very inefficient if the state is large. It does not suffice to return references to all the fields, because the SF is outside the object and so cannot access sub-objects via the references. This is why the SF needs to be able to insert code into the object at runtime.

This solution is not perfect, because a callback can only observe sub-objects of the main object via their methods, and this may not provide access to the information needed by the SF. We might imagine generating meta-observers for every sub-object, but this may be difficult for certain applications. Finally, this approach requires special cooperation from the application, so it will not work for off-the-shelf applications.

6.2.3 Upgrade Database

A scheduling function may need to know the versions and classes of other nodes, e.g., to decide whether its node can upgrade in a round-robin schedule. The upgrade database (UDB) provides a generic, central store for such information. Upgrade layers (ULs) store their node's version and class ID in the UDB after each upgrade and every few minutes (to allow an administrator to monitor the system). SFs can query the UDB to implement globally-coordinated schedules, and the upgrader can query the UDB to monitor upgrade progress. The upgrader can also define additional upgrade-specific tables in the UDB, e.g., a list of nodes that are authorized to upgrade, and can modify these tables to control upgrade progress.

6.2.4 Node Database

In addition to the information in the upgrade database, a scheduling function often needs to know which other nodes it node has communicated with recently. We provide this information as a local database on each node that contains the same kinds of records as the upgrade database for the node's recent peers. Upgrade layers periodically exchange their version and class ID with other ULs and store the information they receive from other ULs. Scheduling functions can query this database for information about recently-contacted nodes (including when each one was last heard from).

6.3 Guidelines

Designing a good scheduling function requires that the upgrader consider several factors. How urgent is the upgrade? How robust is the system to node failures? How well can nodes interoperate via their simulation objects? Are there critical groups of nodes?

As an aid to upgraders, we present basic guidelines for designing scheduling functions. In order of priority, they are:

1. An SF must eventually signal, i.e., its completion must not depend on calls that could fail or deadlock. (An SF can still make such calls, but it must be prepared for them to fail or stall.)
2. An SF should limit service disruption by:
    (a) upgrading nodes that provide redundancy for a service at different times.
    (b) upgrading nodes that provide a new service before upgrading nodes that will use it.
    (c) upgrading nodes that use a deprecated service before upgrading those that provide it.
3. An SF should signal as soon as possible.

We can guarantee that a scheduling function meets the first guideline by limiting the amount of time that it is allowed to run. To this end, we require that each class upgrade definition include not only a scheduling function but also an SF time limit. Choosing this time limit presents its own difficulties, but typically the upgrader can estimate how long an upgrade should take and can use that to choose a conservative time limit.

Alternatively, we could require that each class upgrade definition include a deadline (date and time) and could cause nodes to upgrade immediately when that deadline has passed. This is especially useful if a node is disconnected for a long time and is several upgrades behind, since the expired deadlines will cause it to install the upgrades in rapid succession (regardless of their scheduling functions).

We can relax 2(b) and 2(c) using simulation objects. Future SOs enable non-upgraded nodes to provide new services before they upgrade, so we can ignore 2(b) when we have good future SOs. Past SOs enable upgraded nodes to provide old services, so we can ignore 2(c) when we have good past SOs. In peer-to-peer systems, every node is a server to every other node, so we cannot possibly obey 2(b) or 2(c) when an upgrade adds or removes services. Thus, SOs are vital for upgrading such systems.

Ideally, the system disruption caused by an upgrade would be no more than the expected rate of node restarts in the absence of upgrades. This suggests that we might want an "opportunistic" scheduling function that triggers upgrades when nodes restart on their own, as in proactive recovery [37], or when they quiesce [48]. If we can guarantee that all nodes periodically restart, then this approach obeys guidelines 1 and 2 but not 3.

The reason we have guideline 3 is because we expect that systems will run most efficiently when all the nodes are running the same version, so we want nodes to move to the latest version as soon as possible. But sometimes nodes may continue to run old versions for a long time, e.g., a client may elect to use an old version rather than upgrade. In such cases, the pressure on the client to upgrade will increase over time, as later versions introduce incompatibilities with the old versions that cause calls to the old versions to fail.

Chapter 7

Implementation

This chapter describes Upstart, our prototype implementation of the upgrade infrastructure. Upstart is composed of several parts: the upgrade layer, the upgrade server, the upgrade database, and various supporting scripts and tools. As much as possible, we used existing programs and toolkits to implement these parts. This reduced development time and made the system easier to debug and deploy, since many of the programs we use are installed by default on most systems.

The main challenge in implementing the upgrade infrastructure is making it generic while also making it efficient. Like our approach, most previous approaches to upgrading distributed systems require the ability to control the communication between nodes. To do this efficiently, previous approaches sacrifice generality: they require that users implement both their system and the upgrade components using a particular language and/or middleware system [21, 28, 29, 48, 54, 57, 59, 66, 90, 102]. In contrast, our prototype lets users implement their system, transform functions, and scheduling functions using the languages and tools they prefer. Our only requirement is that nodes in the system communicate by sending messages over sockets.

Achieving good efficiency with this level of flexibility is difficult. The upgrade layer introduces overhead on every message sent or received, so naïve implementations may be impractical for use with the high-performance systems we want to upgrade. We address this challenge by implementing the upgrade layer and simulation objects using event-driven C++. To reduce the burden on upgrade implementors, we provide libraries and code-generation tools that simplify the process of implementing SOs for systems that use Sun RPC [99].

We begin this chapter with a discussion of our design goals and the tradeoffs we considered. We then review our overall architecture and discuss each component in turn. As we go along, we describe the tools and libraries we provide to help users implement upgrades.

7.1 Design Goals and Tradeoffs

We designed our prototype with two goals in mind. First, it should introduce little overhead on system performance—especially when no upgrades are taking place, but also when nodes are communicating via simulation objects. Second, the prototype should support upgrades for systems like NFS [36], SFS [76], Chord/DHash [42, 100], and Thor [71]. These are high-performance, large-scale, data-intensive systems that represent a variety of architectures (client-server and server-to-server) and communication protocols (RPC and message-passing).

Since the upgrade layer intercepts every message, our first goal means we need to implement the upgrade layer in such a way as to minimize its per-message overhead. One approach would be to link the upgrade layer into the application itself, e.g., as a replacement for the RPC library. However, our second goal means there is no one library we can replace: NFS uses Sun RPC via the standard RPC library on Unix, SFS and Chord/DHash use Sun RPC via the SFS asynchronous RPC library, and Thor uses a custom message-passing protocol. Therefore, the upgrade layer must reside at a level that all the applications have in common; we chose to implement it as a dynamically-linked library that intercepts system calls to the socket layer.

Placing the upgrade layer at this low level means our prototype infrastructure can support upgrades for any applications that use sockets. As a result, we can upgrade most off-the-shelf programs. But this also means the upgrade layer must marshal and unmarshal RPCs in order to interpret them, and so incurs more overhead than if we were to intercept at a higher level.

To compensate for this overhead, the upgrade handler and simulation objects are implemented as event-driven C++ objects. This means they run extremely fast and perform well under high I/O load (as is common in the applications of interest). However, event-driven C++ programs can be more difficult to reason about than programs written in type-safe, threaded languages like Java. If we were willing to restrict the set of applications we considered to, e.g., those that communicate via Java RMI [79], then we could implement the upgrade layer as a drop-in replacement for the Java RMI library. This design would avoid the marshaling overhead incurred by our prototype and would make it easier to program the upgrade layer and simulation objects.

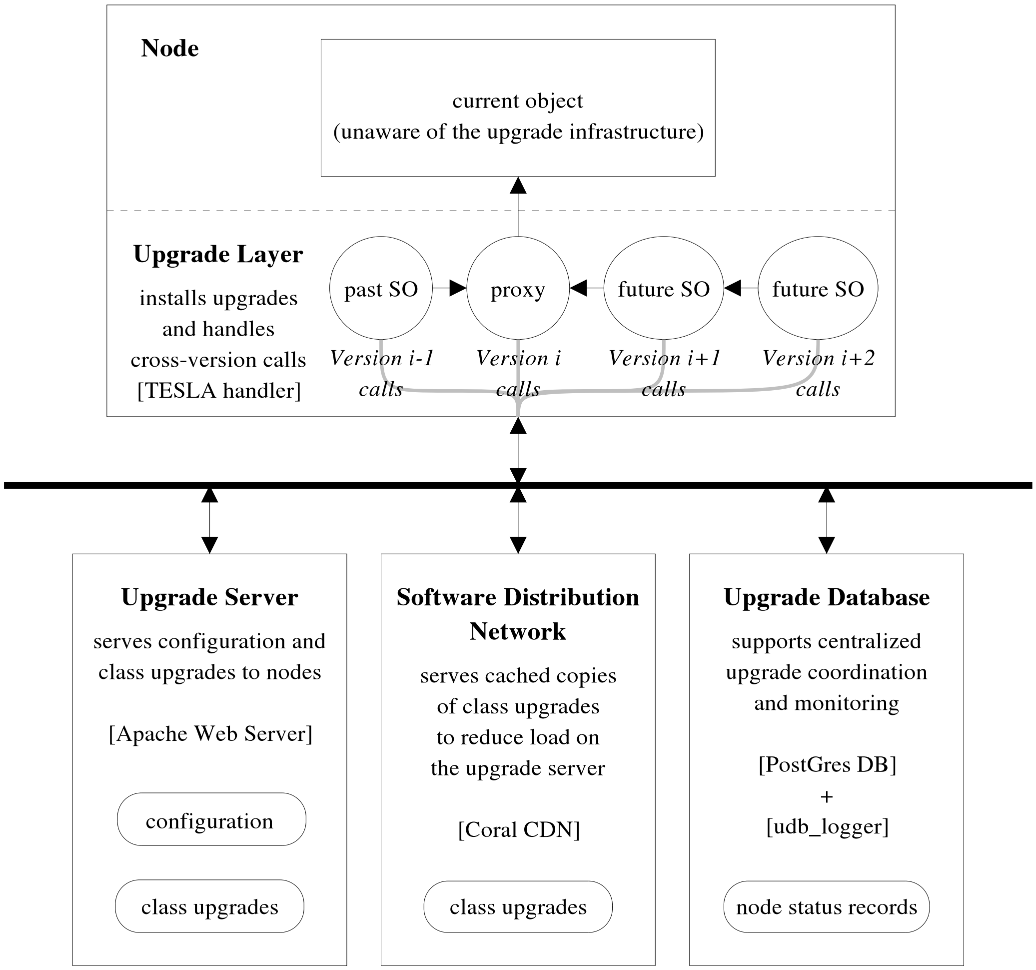


Figure 7-1: *Components of the Upstart prototype*

Thus, our prototype represents an extreme design point: great flexibility at the expense of overhead and programming difficulty. There are many other ways to realize our infrastructure, and other design points will offer different tradeoffs.

7.2 Overview

Figure 7-1 shows how the various parts of the upgrade infrastructure are implemented in Upstart. This chapter is organized as follows. Section 7.3 discusses the upgrade server, software distribution network, and configurations. Section 7.4 covers the upgrade database. Section 7.5 describes how simulation objects are implemented, and Section 7.6, the upgrade layer.

Not all the features discussed in this thesis are implemented in Upstart. We have not implemented filters for class upgrades (Section 2.2), meta-observers (Section 6.2.2), or incremental transform functions (Section 10.3.1). We have not implemented the automatic disallowing described in Section 3.6, nor provided a way for upgraders to designate methods as inexpressible. We have implemented the hybrid model for simulation objects described in Chapter 4 (which combines the interceptor and non-interceptor models), but we have not implemented the reverse-hybrid, delay-hybrid, or notification models.

7.3 Upgrade Server

The main responsibilities of the upgrade server are to store the configuration, class definitions, and upgrade components for the system and make them available for download. The configuration (Section 2.3) is a small file that describes the system's initial schema and class upgrades (it just contains references to the actual class definitions and upgrade components). The class definitions and upgrade components are large binary files or scripts.

Since the configuration, class definitions, and upgrade components are all regular files, we implement the upgrade server as a standard web server (we use Apache [12]). Nodes fetch the files using `wget` [17].

In a system with many nodes, the load of serving these files may be too much for a single server. Class definitions may be large (e.g., several megabytes), so even a few simultaneous downloads can exhaust the bandwidth of a single server. When a new version is announced, all the nodes in the system will attempt to download the new configuration from the upgrade server nearly simultaneously—this is okay, as the configuration is small. But then, all the nodes that are affected by the upgrade will attempt to download their new class definitions from the upgrade server. This means the upgrade server may see sudden bursts of requests for large amounts of data.

Thankfully, the problem of dealing with bursty load is well-studied. Most solutions address the problem by replicating the desired content on several servers and balancing requests for content among those servers [14, 39, 47, 86]. We use the Coral content distribution network [47] to serve downloads for class definitions and upgrade components. We do not use Coral to serve the configuration, because we want nodes to see the latest version of the configuration, not the one cached by the CDN. The configuration is small, so the upgrade server can cache it in memory and can serve many simultaneous downloads easily.

Nodes must be able verify the authenticity of the configuration and class definitions; otherwise, a malicious party could masquerade as the upgrade server and provide false content. We address this by having the upgrader sign each file with its private key using `gpg` [13]. Each node in the system has a copy of the upgrader's public key and can verify the signatures after the download.

Since we require authentication of files, we could instead have served downloads using SSL over HTTP. This works for files downloaded directly from the upgrade server, but not for files downloaded via the CDN. An alternative would be to use a secure network file system [23, 42, 49]. These systems can provide the same load-balancing benefits of a CDN, good security, and an easy-to-use file system interface; but they require the deployment of a special file system client on every node.

7.3.1 Configurations

The configuration of a system is represented as an XML file called `upstart.xml` that resides on the upgrade server. Using XML means that the configuration is human-readable and can be verified automatically for proper syntax. The configuration in Figure 7-2 describes a system with three initial classes of nodes—Web Servers, Doc Servers, and Index Servers—and an upgrade that replaces the first two classes. The attributes of each element (e.g., `newclass`, `library`) correspond to those described in Appendix A.

7.3.2 `upcheck`

The `upcheck` utility verifies that a configuration has the correct syntax and satisfies various sanity checks. It checks the structure of the XML using the Upstart document type declaration (DTD) and checks the semantic constraints described in Appendix A (e.g., a subtype upgrade defines a future SO). For each field that names a file, `upcheck` checks that the file exists on the local file system, has the proper permissions, and is digitally signed. The utility also reports the set of classes that are defined for each version and complains if the an upgrade has an invalid `oldclass` (i.e., a class that does not exist in the schema preceding the upgrade). Figure 7-3 gives the output of `upcheck` for the configuration in Figure 7-2 when `upstart.xml` has a bad signature and `randomized.sh` is missing.

```
<!DOCTYPE config SYSTEM "upstart.dtd">
<config number="1">
  <initial newclass="WebServer"
           library="libWebServer.so"
           code="WebServer.tar.gz"/>
  <initial newclass="DocServer"
           library="libDocServer.so"
           code="DocServer.tar.gz"/>
  <initial newclass="IndexServer"
           library="libIndexServer.so"
           code="IndexServer.tar.gz"/>
  <version number="2">
    <upgrade oldclass="WebServer"
             newclass="TWebServer"
             code="TWebServer.tar.gz"
             type="sametype"
             library="libWebServerSim.so"
             sf="roundrobin.sh"
             sfMaxSecs="60"
             tf="wstf.sh"/>
    <upgrade oldclass="DocServer"
             newclass="TDocServer"
             code="TDocServer.tar.gz"
             type="subtype"
             library="libDocServerSim.so"
             sf="randomized.sh"
             sfMaxSecs="30"
             tf="dstf.sh"/>
  </version>
</config>
```

Figure 7-2: *A configuration file*

```
* signature 'upstart.xml.sig' does not exist or is invalid;
  create it with: gpg --yes -b upstart.xml
- version 1 classes are WebServer, DocServer, IndexServer
- libDocServerSim.so must define a future SO for version 2
* file 'randomized.sh' does not exist or is not executable
- version 2 classes are IndexServer, TWebServer, TDocServer
- done
```

Figure 7-3: *Output of* upcheck*. Lines preceded by '-' provide information; lines preceded by '*' report errors.*

7.4 Upgrade Database

The upgrade database (UDB) provides a central store for information about the state of the system: the upgrade status of individual nodes and tables indicating which nodes are allowed to upgrade. We have implemented the UDB as a PostGres database that resides on the upgrade server [101].

The `headers` table in the UDB contains a record for each node that contains its IP address, its class ID, its current version, the minimum and maximum version it supports, and a timestamp indicating when the record was added. Nodes insert new records in the UDB periodically or whenever this information changes (e.g., because a node learns of a new version).

New records for a node do not overwrite its old records in the UDB. This way, we have a full trace of each node's availability and upgrade status over time. Of course, we cannot guarantee that every node will have up-to-date records in the UDB, because updates may be lost due to node failures or communication problems. But nodes that are up and have access to the UDB will eventually insert new records. A background process can discard or archive old records periodically, e.g. once a day or once a week.

Nodes do not write to the UDB directly, because this would cause too much contention in a large system. Instead, nodes send their header over UDP to a `udb_logger` process running on the upgrade server that in turn inserts records in the UDB. Under heavy load, some headers may be lost; but this is okay, since they will be sent again later.

A malicious party could attempt a denial-of-service attack against the UDB by flooding it with headers, thus preventing legitimate nodes from reporting or exhausting the space in the UDB. We can protect against this attack by limiting the rate at which the `udb_logger` accepts new records for each node. This scheme requires that `udb_logger` keep a small amount of state for each node in the system.

A malicious party could also attempt to corrupt the data in the UDB by sending false headers, e.g., headers that report incorrect classes or versions for nodes. We could protect against this attack by requiring that nodes sign their headers with their private key. Then, the attacker must compromise a node's private key to create false headers for it. This scheme requires that `udb_logger` have the public key for each node and verify the signature for each header.

7.5 Simulation Objects

Chapter 4 discussed several models for how to use simulation objects. In Upstart, we have implemented the hybrid model (Section 4.4). We have not implemented the variants of the hybrid model or the notification model, because these models were developed after our prototype was complete. We expect implementing these other models to be straightforward, because they just change how nodes initialize SOs and dispatch calls.

A node does not necessarily have an object for every version. We expect most new versions to define class upgrades for just one or two old classes, so nodes whose classes are unaffected by the new version simply direct calls for the new version to the object that handles the previous version. We say that a node *skips* the versions that do not affect it.

An important feature of the hybrid model is that future SOs for incompatible upgrades can run as interceptors, which means they must handle calls for both the old and new types. These SOs need a way to distinguish between these calls, since there may be name conflicts. Our solution is for the upgrader to actually provide two objects: a *bridge* that intercepts for the old object and a *renamer* that intercepts for the new object. The bridge does the real work: it implements the old and new specifications, but it may rename the new methods to avoid conflicts. The renamer delegates calls for new methods to the appropriate (renamed) methods on the bridge. Of course, we could simplify this process by generating the renamer automatically from a renaming map.

Constructing the SO chains and dispatch tables for a node is non-trivial, given that a single node may have skipped versions, bridges, interceptors, and non-interceptors all at once. But the algorithm to do this is reasonably straightforward, and we provide it as pseudocode in Appendix B.

7.5.1 Programming SOs

Simulation objects are implemented as C++ objects; Figure 7-4 presents the `SO` interface. This interface is minimal; it just specifies the `from_net` method, which is how the SO receives data from the network. An SO must parse the data it receives into messages and handle them as required by its specification.

In practice, a simulation object implements a subclass of `SO` that provides a richer interface for other SOs than `from_net`. In our prototype, we focus on applications that use Sun RPC [99] to communicate, so our SOs implement a subclass of `rpcSO`, which is given in Figure 7-5 along with its superclass, `rpcObj`. `rpcSO` is constructed with two parameters: an `rpc_program` (a runtime

```
// a version number
typedef uint32_t versno_t;

// a callback for communicating with the network or application
typedef callback<void, ref<address>, data>::ref netcb_t;

class SO
{
public:
    SO();
    virtual ~SO();
    // called when data arrives from the network
    virtual void from_net(netcb_t to_src, ref<address> from, data d) = 0;
    virtual void *getThis() = 0; // needed to make downcasts work with DLLs
};
```

Figure 7-4: *C++ signature for SOs*

representation of a Sun RPC interface specification produced by `rpcc` [76]) and a flag indicating whether the network transport uses reliable streams or unreliable datagrams (this is needed to marshal RPCs correctly). `rpcSO` parses the data it receives via `from_net` into RPCs and invokes `call` for each RPC with the appropriate transaction ID (`xid`), procedure number, argument, and continuation (which returns the reply to the caller). The SO implementor need only provide an implementation for `call`.

The upgrade layer constructs an SO by calling a factory procedure defined in the dynamically-linked library (DLL) provided with the class upgrade for that SO. Depending on the type of the upgrade, the DLL may include up to three such factory procedures, `createPastSO`, `createFutureSO` and `createBridge`. The signatures for these procedures are given in Figure 7-6; `createBridge` has the same signature as `createFutureSO`.

The upgrade layer passes the factory procedure a reference to the SO's delegate and, in the case of future SOs and bridges, a flag indicating whether the SO is running as an interceptor. The reference to the delegate is statically typed as `SO*`, but in the case of `rpcSO`, it is downcast to `rpcObj*`. This allows the SO implementor to invoke the `call` method of the delegate directly.

The past SO and future SO closest to the current object cannot call methods of the current object directly, because the SOs and the current object run in separate processes, as depicted in Figure 7-7. Those SOs delegate to a *proxy* object that implements the interface they expect (e.g., `rpcObj`). A proxy has several responsibilities:

```
// a callback for returning method results
typedef callback<void, void *, clnt_stat>::ref rescb_t;

class rpcObj
{
public:
    rpcObj(const rpc_program &p, const bool isstr);
    virtual ~rpcObj();
    virtual void from_net(netcb_t to_src, ref<address> from, data d);
    virtual void call(xid_t xid, ref<address> from,
                      procno_t proc, void *arg, rescb_t cb) = 0;
};

class rpcSO : public rpcObj, public SO
{
public:
    rpcSO(const rpc_program &p, const bool isstr);
    virtual ~rpcSO();
};
```

Figure 7-5: *C++ signature for Sun RPC SOs*

```
// creates the SO for a given version
// given a pointer to the next object in the chain
typedef SO *createFutureSO_t(const versno_t vers, const bool isstr,
                             SO *next, bool interceptor);
typedef SO *createPastSO_t(const versno_t vers, const bool isstr,
                           SO *next);
```

Figure 7-6: *Factory procedures for creating simulation objects*

- for calls received via the network: forward these calls to the application, and when the application replies to these calls, forward the reply on to the original caller via the network.
- for calls received via its methods: forward these calls to the application, and when the application replies to these calls, send the reply to the object that made the call (via its continuation).
- for outgoing calls made by the application: forward these calls to the network, and when the receiver replies to these calls, send the reply on to the application.

Managing these tasks requires some care; thankfully, we can implement proxies without any user-defined code. We provide a class called `rpcProxy` (Figure 7-8) that implements these tasks for Sun RPC given the appropriate `rpc_program`. `rpcProxy` implements the `Proxy` interface (Figure 7-9), which is what the upgrade layer uses to pass data from the network and from the application to

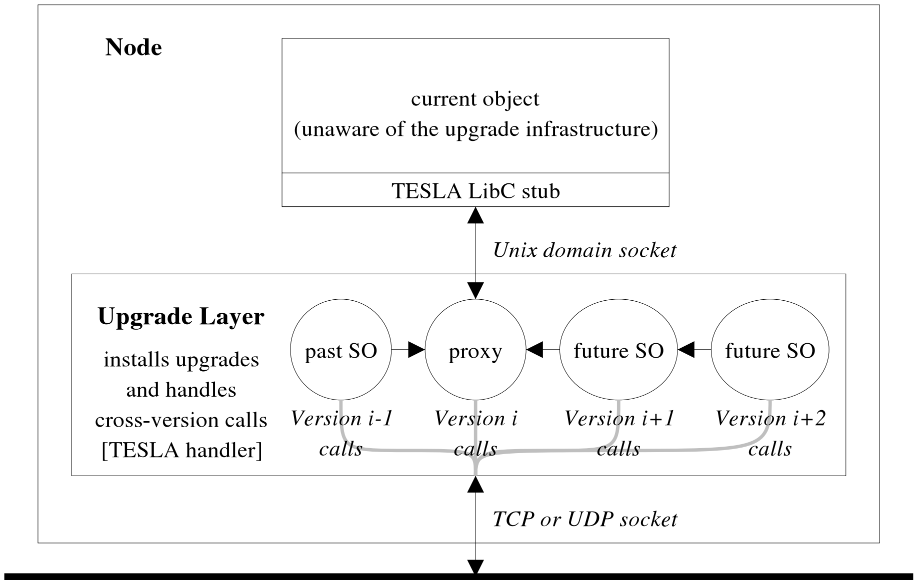


Figure 7-7: *Process structure of the upgrade layer*

```
class rpcProxy : public rpcObj, public Proxy
{
public:
    rpcProxy(const rpc_program &p, const bool isstr,
             netcb_t to_net, netcb_t to_app);
    virtual ~rpcProxy();
    virtual void from_app(ref<address> to, data d);
};
```

Figure 7-8: *C++ signature for Sun RPC proxies*

the proxy. The upgrade layer constructs proxies by calling the `createProxy` factory procedure (Figure 7-10), which is defined in the DLL provided by the upgrader for the current class.

7.6 Upgrade Layer

The upgrade layer runs on each node and is responsible for adding version numbers to outgoing messages, stripping version numbers from incoming messages, dispatching incoming messages to the appropriate proxy or simulation object, allowing the proxy to communicate with the application, exchanging header information between nodes, and checking for, downloading, and installing new upgrades.

```
class Proxy : public SO
{
public:
    Proxy();
    virtual ~Proxy();
    // called when data arrives from the application
    virtual void from_app(ref<address> to, data d) = 0;
};
```

Figure 7-9: *C++ signature for proxies*

```
// creates the proxy for a given version
// given callbacks to communicate with the network and the application
typedef Proxy *createProxy_t(const versno_t vers, const bool isstr,
                             netcb_t to_net, netcb_t to_app);
```

Figure 7-10: *Factory procedure for creating proxies*

We decompose this behavior into two modules: the *upgrade manager* downloads and installs new upgrades, and the *upgrade handler* manages messages and dispatching. Both modules run in the same process and share state directly, but they run in a separate process from the application, as depicted in Figure 7-7. This separation is important: if the application has a bug (e.g, that causes it to loop forever), the upgrade manager must be able to make progress so that it can download and install code that fixes the bug.

7.6.1 Upgrade Handler

The upgrade handler is implemented as a TESLA handler [92]. TESLA is a dynamic interposition library that intercepts `socket`, `read`, and `write` calls made by an application and redirects them to *handler* objects. TESLA handlers can transform these calls to enhance the communication layer, e.g., by encrypting messages or supporting session migration. TESLA also supports composing multiple handlers, but we do not use this feature.

We initially considered implementing the upgrade handler as an explicit TCP proxy, i.e., a process that listens on the application's port and forwards calls to the application itself, which listens on a private port. This design fails for applications that exchange their address (host and port) with other nodes, because the application will advertise its private port instead of the proxy port. Peer-to-peer systems like Chord [100] and Gnutella [15] are common examples of such applications.

TESLA is transparent to the application, so the application can listen on its usual port and communicate normally. When the application creates a new socket, TESLA creates an instance of the upgrade handler and provides an interface that allows the handler to write data to the network or to the application. When the application writes data to the socket or when data arrives on that socket from the network, TESLA notifies the upgrade handler via method calls.

Protocol

The upgrade layer receives raw data from the application and from the network. It does not know how to parse this data into messages, nor does it need to: as discussed in section 7.5, the proxy and simulation objects handle the marshaling and unmarshaling of messages. This means the upgrade layer just needs to know which version number to attach to outgoing messages, and to which object to dispatch incoming messages.

The upgrade layer associates a version number with the proxy and each simulation object. When one of the objects writes a message to the network, the upgrade layer prepends that version number to the message. The upgrade layer encodes message boundaries by prepending a length to each message. Thus, the common-case overhead is 8 bytes (two 4-byte integers) per message.

When the upgrade layer receives data from the network, it reads the message length, the version number, and the message itself. The upgrade layer can handle calls for any version between *minv* and *maxv* (inclusive), and it must dispatch each call to the appropriate object (past SO, future SO, or proxy). An SO may, in turn, make calls on the object to which it delegates. If the version number falls outside the supported versions, the upgrade layer discards the message and replies to the caller with a small error code.

Running a node

When the software of a node is started for the first time, the upgrade layer needs to know the location of the upgrade server and the initial class of the node. We assume the upgrade database is on the same host as the upgrade server. We provide a script called `upstart` that takes care of the node setup:

```
upstart host path classID
```

This invocation tells `upstart` that the upgrade server and upgrade database are on host `host`, that the configuration and node software is in the directory `path` on that host, and that the initial class of this node is `classID`. For example,

```
upstart banjo.lcs.mit.edu /space/upstart/chord Chord
```

tells `upstart` to initialize this node as class `Chord` from the upgrade server on host `banjo.lcs.mit.edu` and path `/space/upstart/chord`. `upstart` copies the configuration (`upstart.xml`) from the upgrade server, looks up the initializer for class `Chord` in the configuration, downloads and installs the code and library for class `Chord`, and starts the node software. `upstart` assumes that the code for the node includes an executable called `start` that actually starts and recovers the node software; `start` is usually a script that passes the appropriate command-line arguments to the actual executable.

After the initial setup, the node can be restarted by invoking `upstart` with no arguments (`upstart` saves its state in a local file called `state`). In this case, `upstart` first invokes `upstop` to shut down the node software, then checks whether any transforms are pending (as discussed in Section 7.6.2), and finally restarts the node software.

`upstop` attempts to terminate the node software by invoking a user-provided executable called `stop`. If this fails, `upstop` sends `SIGTERM` and finally `SIGKILL` to the top-level process of the node software. This is guaranteed to kill the top-level process, but in the case of multi-process systems, some other processes may remain.

Once the node is running, it will keep running until it crashes or it is explicitly terminated with the command `upstop`. If the node crashes and the `UPSTART_RESTART` environment variable is set, the upgrade layer will automatically invoke `upstart` to restart the node. This can help a node tolerate buggy software until an upgrade is available.

7.6.2 Upgrade Manager

The upgrade manager is responsible for periodically polling the upgrade server for new configurations, downloading new upgrades and libraries, installing future SOs on-the-fly, running scheduling functions, installing new software, and running transform functions. With the exception of installing new future SOs, the upgrade manager accomplishes these tasks by spawning processes to handle them (`wget` and `gpg` for downloads and the appropriate scripts for SFs and TFs). We discuss some of the details in this section.

The upgrade manager maintains a periodic timer that causes it to poll the upgrade server every few minutes. But when the upgrade handler hears about a version higher than its *maxv*, it causes the upgrade manager to poll immediately. This ensures rapid dissemination of new versions and minimizes the time when the future SO is not yet installed. So that it does not poll too often, the manager maintains a minimum polling period of five seconds. In a large system, this minimum should be increased to limit the number of nodes that attempt to poll the upgrade server at the same time.

The upgrade manager also causes the upgrade handler to piggyback headers on the messages it sends to other nodes. A header includes the minimum, maximum, and current version of the node and its class ID, so it is too large to include with every message. Instead, the manager maintains a record of when it last sent a header to each node and causes the manager to piggyback a header once per minute. This ensures that the overhead of header exchange stays constant, regardless of the rate at which nodes communicate.

The manager keeps a local database (LDB) of the headers it has received from other nodes; scheduling functions can query the LDB to make local scheduling decisions. We have implemented the LDB as a PostGres database [101]; it has the same `headers` table as the UDB, except the LDB only contains headers from nodes that have contacted the local node.

Writing headers to the LDB whenever they are received would be time-consuming and would affect the critical path of each message; instead, the upgrade manager forwards headers over UDP to a local `udb_logger` process that runs in the background and stores the headers in the LDB (much like `syslogd`).

Scheduling Functions

A scheduling function is implemented as a program that runs in a separate process from the node software. SFs can access the persistent state of the node directly (read-only), and they can access the volatile state of the node via its observers. As we discussed in Section 6.2.2, we can provide the SF with privileged access to the volatile state using meta-observers, but we have not implemented this yet.

Scheduling functions can query the LDB directly using SQL. SFs can query the central upgrade database using SQL over SSH or a database-access protocol like ODBC.

Transform Functions and Node Recovery

A transform function is implemented as a program that runs in its own process. The upgrade manager indicates that a transform is pending by writing the node's new version, new class ID, and an end marker to a file called `upstart.tf`. The manager then invokes `upstart`, which in turn shuts down the node software, reads this file, invokes the TF for the indicated upgrade, records the new version and class ID, removes `upstart.tf`, and starts and recovers the new node software (by invoking `start`). It is the TF's responsibility to run the old class's recover method to clean up the persistent state (if necessary). If the node fails while the TF is running, it can simply invoke `upstart` again when it recovers. `upstart` will restart the transform if `upstart.tf` is still there, otherwise it knows the transform completed and will just start the node software.

Chapter 8

Evaluation

When evaluating our upgrade infrastructure, we might hope to answer questions like:

- How long does an upgrade take?
- How disruptive is its upgrade schedule?
- How quickly does its transform run?
- How well does its simulation object work?

But every upgrade is different, and the answers to these questions depend on the system being upgraded and the nature of the upgrade itself. Furthermore, upgrades are rare, so whether a transform takes one minute or two matters little in the common-case.

In quantitative terms, what really matters is the overhead imposed by the upgrade layer on application performance when no upgrades are happening, as this is the common case. We also want to confirm that the additional overhead imposed by running simulation objects is small, since otherwise we will never be willing to use them. Section 8.1 presents experiments that measure these overheads and show them to be reasonable.

We are also interested in a qualitative evaluation of how well the upgrade system actually works. Section 8.2 discusses some of the upgrades we have actually implemented and run.

8.1 Overhead

To understand the overhead of our prototype, we measured the performance of various applications in several scenarios. The *Baseline* scenario measures the performance of the application alone. The

TESLA scenario measures the performance of the application running with the TESLA "dummy" handler on all nodes. The difference between *TESLA* and *Baseline* is the overhead for interposing between the application and the socket layer and for context switching between the application and the TESLA process.

The *Upstart* scenario measures the performance of the application running with the upgrade layer on all nodes. The difference between *Upstart* and *TESLA* is the overhead for labeling messages with version numbers, exchanging headers, and—most importantly—data copying. Each outgoing message is copied to a new buffer so that a version number can be prepended to it; this could be avoided using scatter-gather I/O, but TESLA doesn't support this. TCP communication is buffered by `rpcProxy` and `rpcSO` so that they can reassemble fragmented RPC messages. While this second buffer is unavoidable (due to RPC semantics), the first is unnecessary and could be removed by changing TESLA to support scatter-gather I/O. This optimization is future work.

The final scenario, *With SO*, measures the performance of the application with a null SO on the server, i.e., an SO that just delegates. The difference between *With SO* and *Upstart* is the additional overhead of dispatching calls through the SO, unmarshaling data from the network into RPCs, and marshaling RPCs back to the network (to pass to the application). Calls to SOs may also be slower than normal calls, because SOs are dynamically-loaded objects, so the compiler is unable to optimize calls to SOs as well as it can optimize statically-linked code.

We measure the performance of three applications in each of the scenarios listed above. The applications have very different communication patterns. In *Null RPC*, a client issues small remote procedure calls to a server one-at-a-time. In *TCP data transfer*, a client transfers a bulk data to a server using TCP (no RPCs). In *DHash block fetch*, a client retrieves data blocks from a distributed hash table composed of several servers; each fetch operation is composed of several small LOOKUP RPCs issued sequentially, then several large FETCH RPCs issued in parallel.

Finally, we run these experiments both on a local gigabit ethernet (transfer rates of up to 125 MB/s) and on the Internet. The local network achieves very high application performance and so demonstrates the worst-case overhead.

8.1.1 Null RPC

We measure the latency of null remote procedure calls, i.e., RPCs that have no arguments or return values. A single client issues RPCs synchronously to a single server. This application is extremely lightweight, so this experiment measures the worst-case overhead for using the upgrade layer.

Figure 8-1 shows the latencies of null RPCs on a gigabit ethernet. The upper graph has a box plot for each scenario: each box encloses the middle 50% of latencies for that scenario, and the vertical lines extend from the minimum to the maximum latency. The dotted line connects the median latencies of the scenarios.

The lower graph plots the cumulative distribution function (CDF) for each scenario. The y-value is the fraction of latencies whose value is less than the x-value. Thus, a horizontal line at $y = 0.5$ intercepts each curve at the median value for the corresponding scenario.

The median latency of *TESLA* is 21% greater than *Baseline*; *Upstart* is 205% greater than *Baseline*; and *With SO* is 208% greater than *Baseline*. The overhead in the *Upstart* case is likely due to the fact that every outgoing RPC is buffered in the upgrade layer; if so, this overhead could be removed with optimization. The additional overhead of *With SO* is relatively small; it is due to the additional dispatching, marshaling, and unmarshaling done to pass the RPC through the SO. The CDF provides more information: most of the *Baseline* and *TESLA* latencies are around 400μs, while most of the *Upstart* and *With SO* latencies are around 800μs. This suggests that the data copying in the upgrade layer imposes a constant overhead of around 400μs, and the additional overhead imposed by running with an SO is small.

The Internet experiments show that the overhead of the upgrade layer disappears in a high-latency network. Figure 8-2 shows the latencies of null RPCs from a client at UC San Diego (UCSD) to a server at MIT. The median latency of *With SO* is 1.1% greater than *Baseline*, and the other scenarios are all within 1% of *Baseline*.

8.1.2 TCP data transfer

We measure the latency of bulk data movement from a single client to a single server using TCP. We are interested in how the throughput (data transferred per unit time) is affected by using the upgrade layer.

Figure 8-3 shows the latencies of transferring 100 MB on a gigabit ethernet. The median *Baseline* throughput is 110 MB/s; *TESLA*, 109 MB/s; and *Upstart*, 70 MB/s. This overhead is likely due to the buffering done in the upgrade layer and, if so, could be removed with optimization.

Figure 8-4 shows the latencies of transferring 1 MB on the Internet. In these experiments, the server is located at MIT, and the client is located at UCSD. The median *Baseline* throughput is 378 KB/s; *TESLA*, 377 KB/s; *Upstart*, 392 MB/s. We believe this strange (but repeatable) increase in throughput comes from an artifact in TESLA and the upgrade layer that prevents TCP from

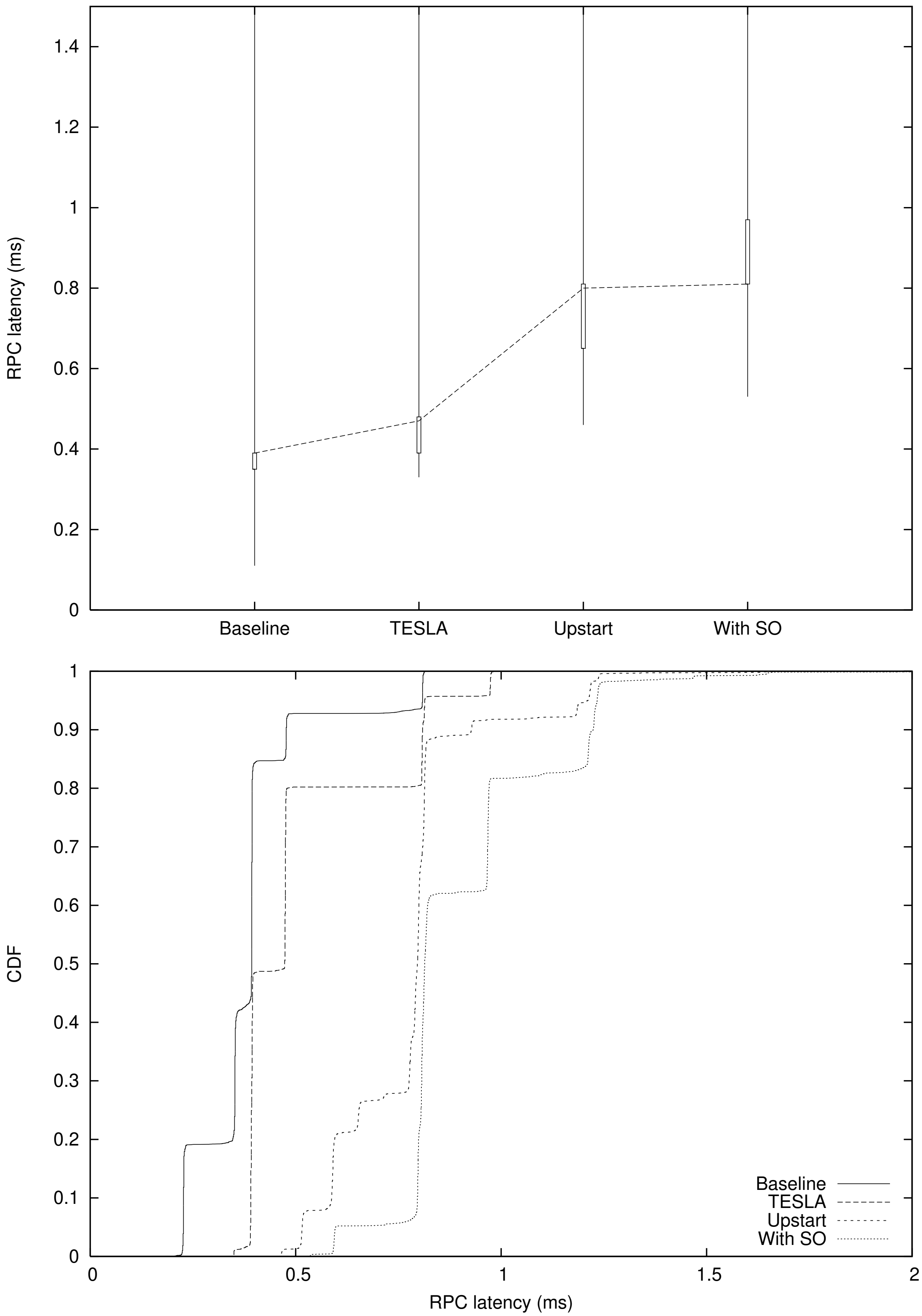


Figure 8-1: *Time to do a null RPC on a gigabit LAN (N=10000)*

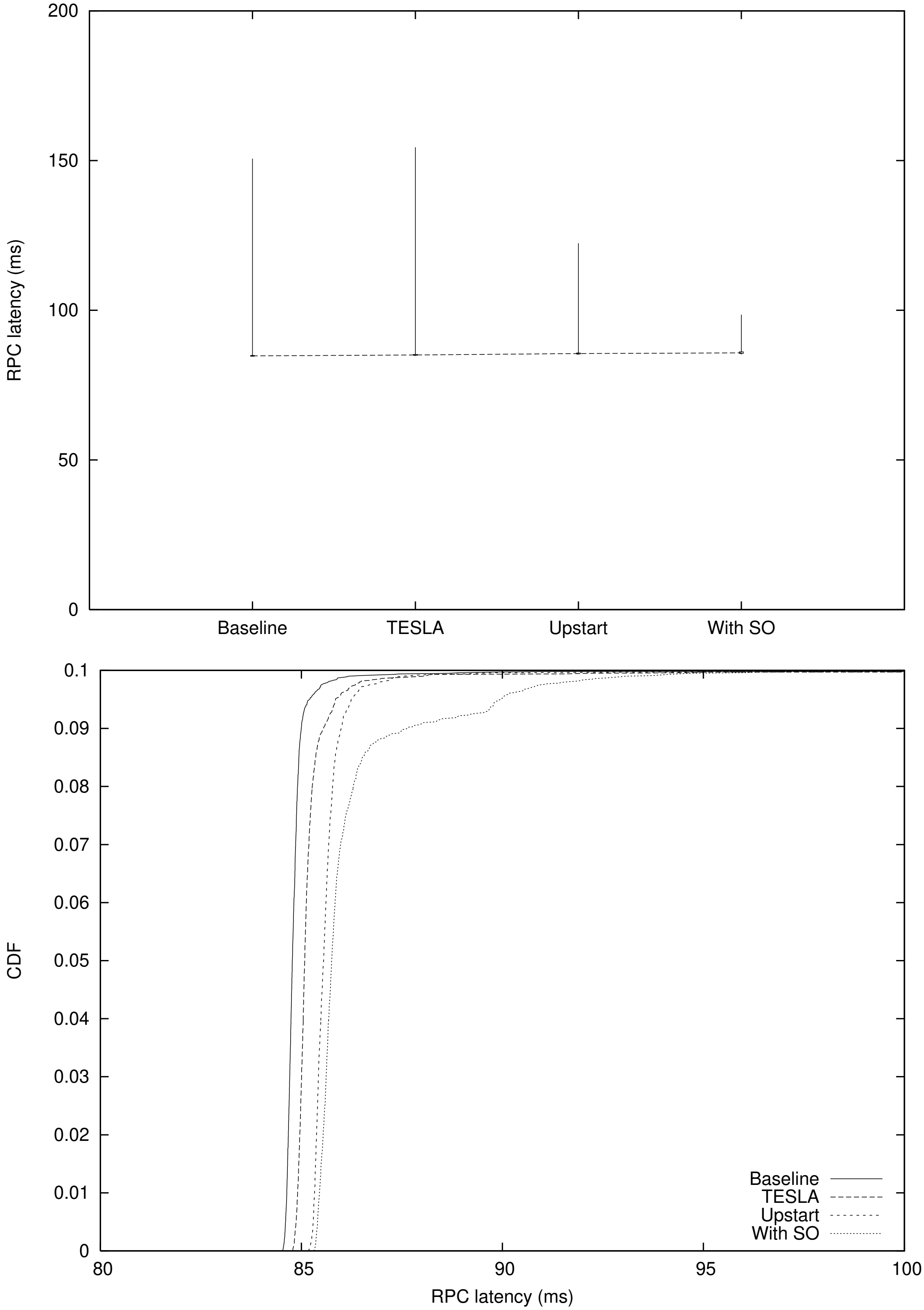


Figure 8-2: *Time to do a null RPC from MIT to UC San Diego (N=10000)*

using an over-large flow control window and so enables it to avoid packet loss and achieve higher throughput.

8.1.3 DHash block fetch

We measure the latency of fetching blocks from DHash. DHash is a peer-to-peer application that implements a distributed hash table, i.e., a key-value store where different servers store the values for different keys. The values are 8 KB blocks that are stored as 1 KB erasure-coded fragments, which means a client actually downloads a block from several servers in parallel. A client locates the fragments of block by looking up the servers responsible for that key, which involves issuing several small LOOKUP RPCs to various servers. The client then issues FETCH RPCs in parallel to these servers; the return values of the FETCH RPCs are the 1 KB fragments.

Our experiments use a DHash system composed of four servers and a single client that resides on one of the servers. Each server runs 14 virtual servers, for a total network size of 56 virtual servers. We store 256 8 KB blocks of random data in the system and measure how long the client takes to fetch each block (one-at-a-time).

Figure 8-5 shows the latencies of DHash fetches on a gigabit ethernet. The median latency of *TESLA* is 22% greater than *Baseline*; and *Upstart* is 29% greater than *Baseline*. Thus, the bulk of the overhead in this experiment comes from using TESLA, not the upgrade layer. This is probably due to context switching and data transfer between the TESLA process and DHash.

Figure 8-6 shows the latencies of DHash fetches on the Internet. In these experiments, the four servers are located at MIT, UCSD, Denmark, and Taiwan, and the client is located at MIT. The median latency of *TESLA* is 7.8% greater than *Baseline*; and *Upstart* is 12% greater than *Baseline*. The overhead of TESLA and Upstart in the Internet is less than in the LAN, but it is still significant. Again, this is likely due to data copying.

8.1.4 Summary

We conclude that the overhead of our prototype should be acceptable for many applications, but it may be too much for applications that require very high throughput. Optimization may fix this, but in these more demanding environments, it may also make sense to use a specialized upgrade layer that interposes at a higher level than our prototype does. By raising the level of interposition, we can eliminate the overhead of extra data copying, context switches, and redundant RPC marshaling and unmarshaling.

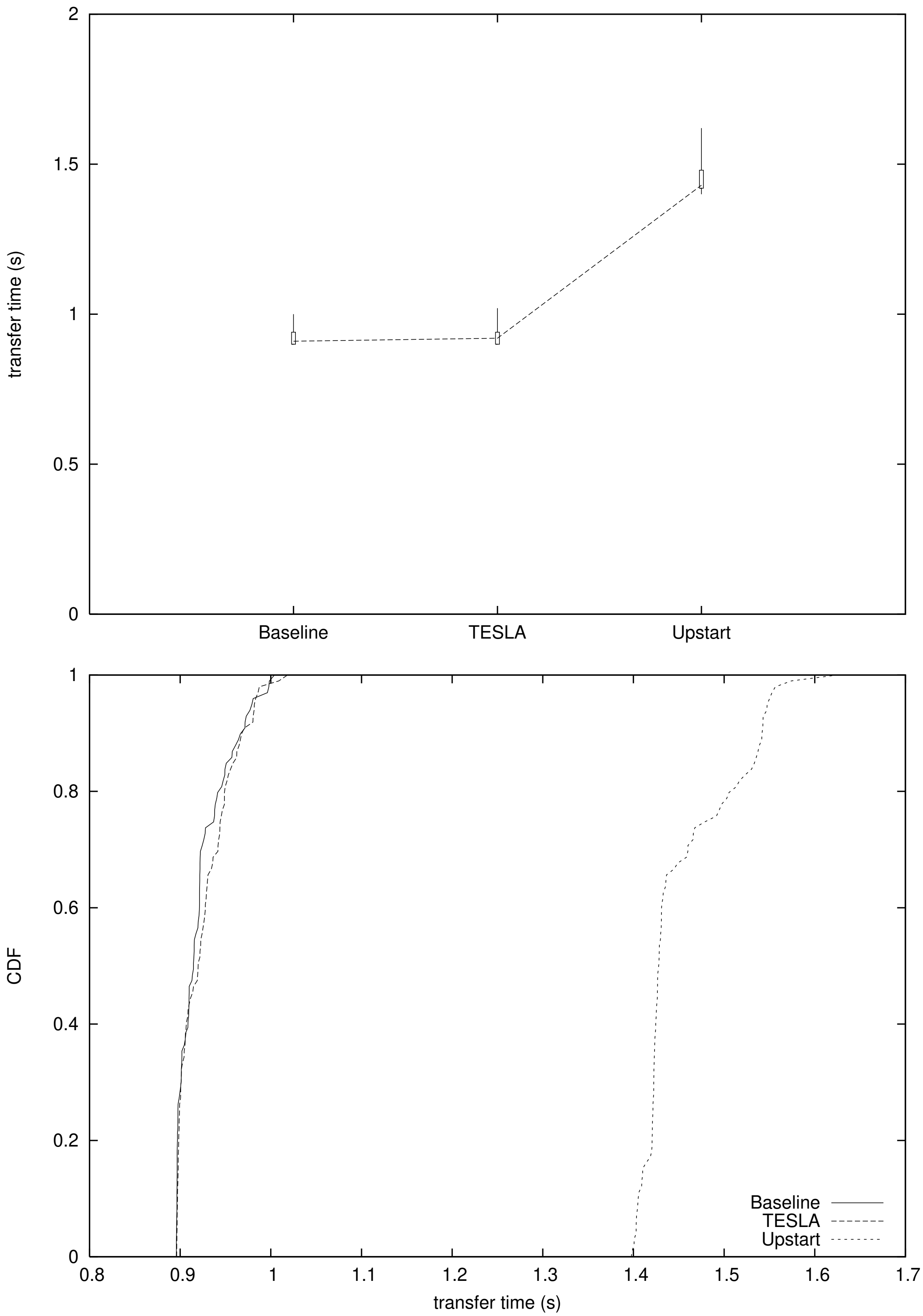


Figure 8-3: *Time to transfer 100 MB on a gigabit LAN (N=100)*

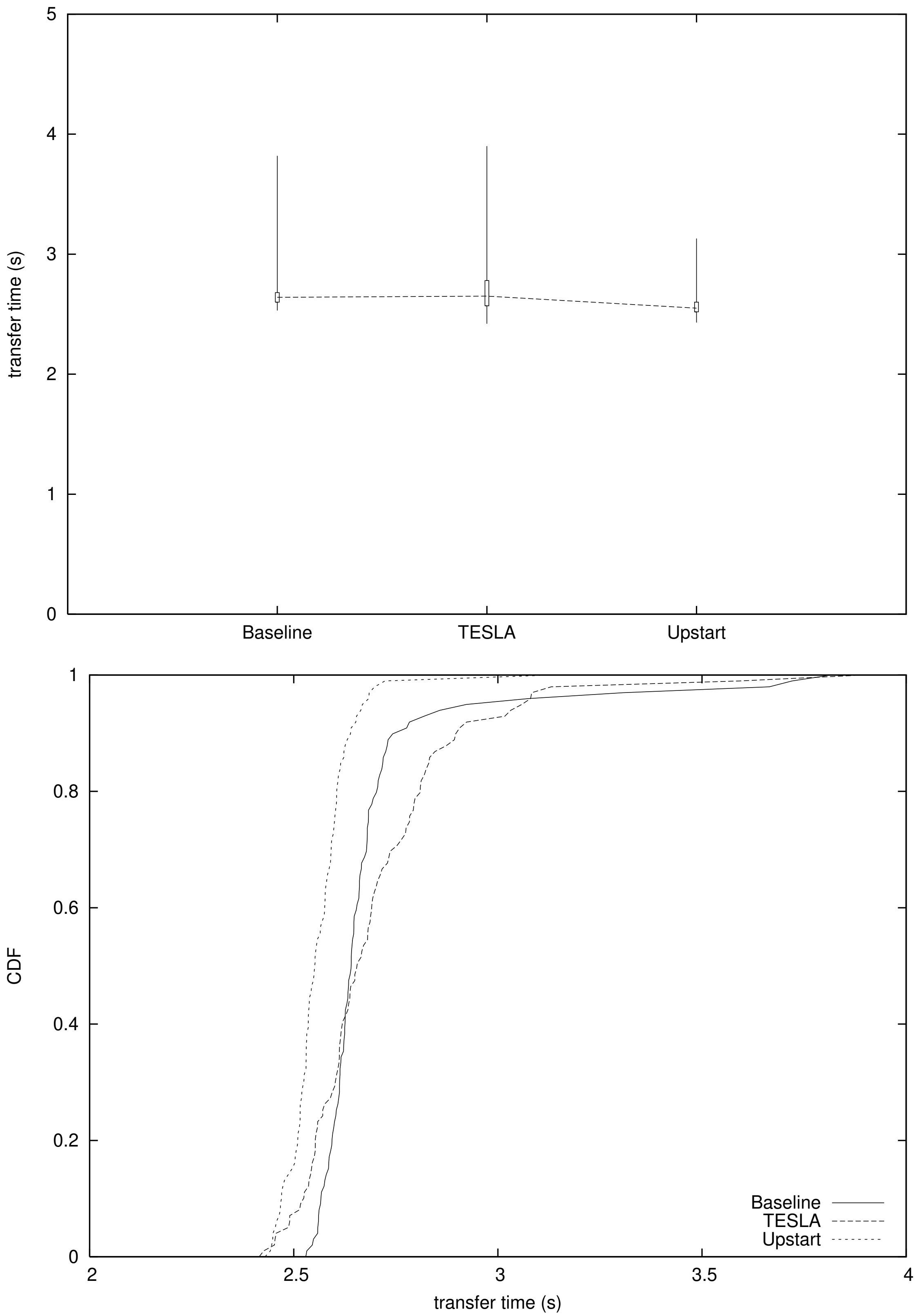


Figure 8-4: *Time to transfer 1 MB from MIT to UC San Diego (N=100)*

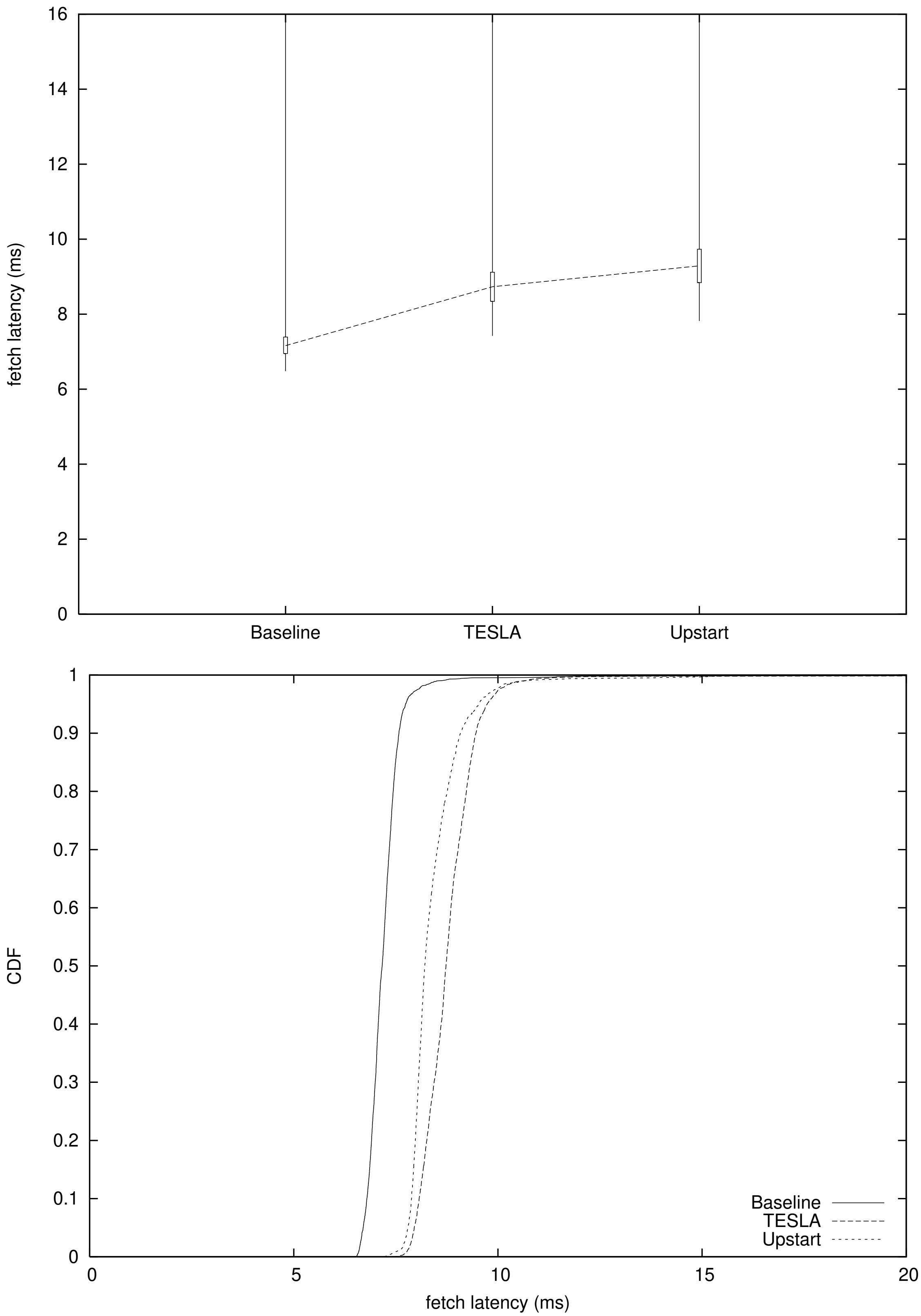


Figure 8-5: *Time to fetch an 8 KB block from DHash on a gigabit LAN (N=768)*

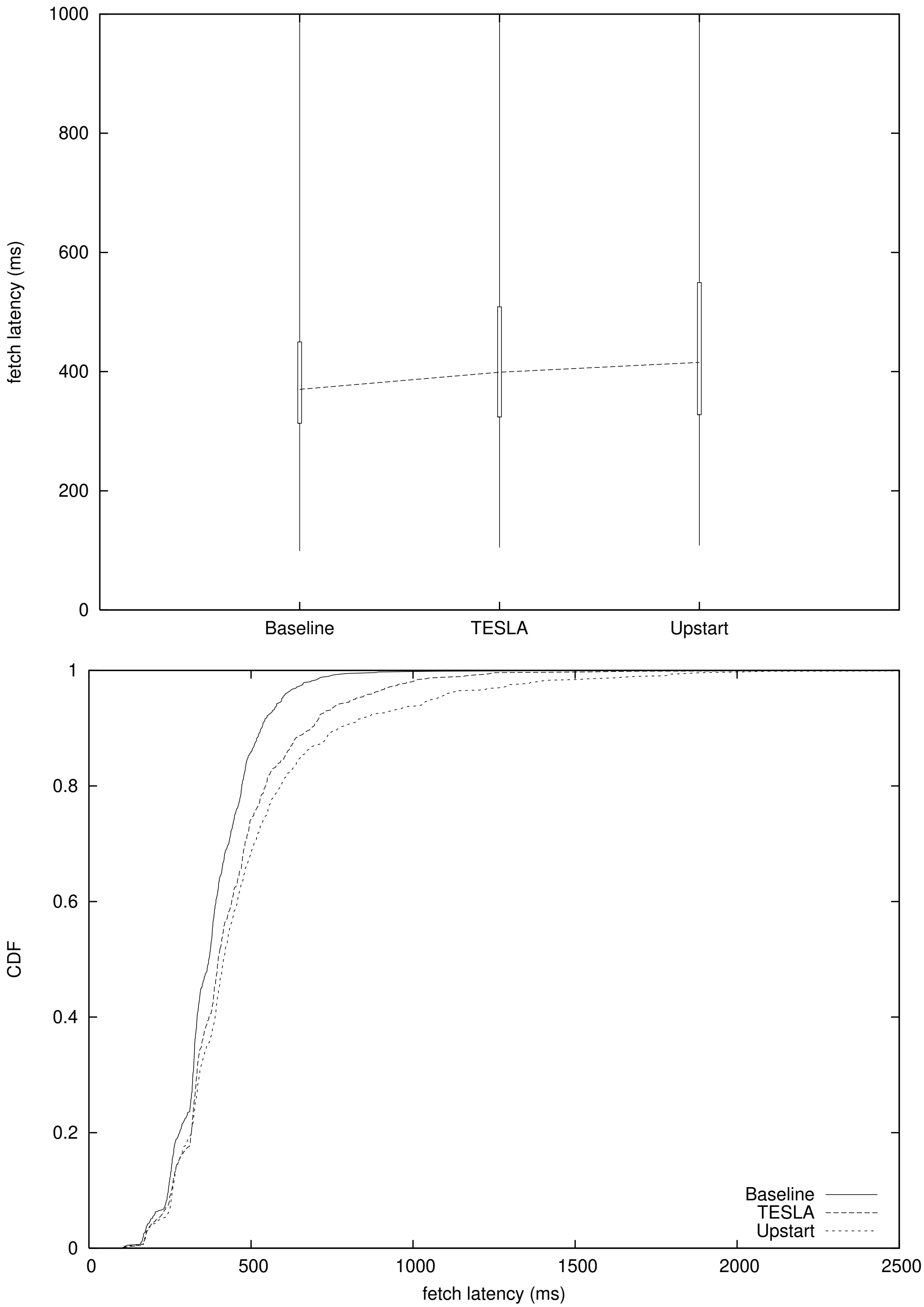


Figure 8-6: *Time to fetch an 8 KB block from DHash on the Internet (N=768)*

8.2 Qualitative Results

Few upgrades actually require complex scheduling functions, transform functions, and simulation objects simultaneously. Rather than try to create one gigantic upgrade that exercises all these components, we evaluate each component separately using simple upgrades.

8.2.1 Simulation Objects

To evaluate the difficulty of writing simulation objects, we created an upgrade for a small application called DocServer. DocServer stores a mapping from names to "documents," which are just strings. This mapping is persistent state; DocServer stores it in a local database. Clients can call methods on the server to add new documents, replace existing documents, and fetch documents by name.

The upgrade we implemented allows clients to add comments to documents. When clients request a document's contents, the comments are automatically appended to it. Like the documents, comments are persistent state.

The interesting parts of this upgrade are the simulation objects. The new version introduces new state to store the comments, and the future SO must maintain this state before the upgrade occurs. Our implementation of the future SO keeps a mapping from document names to comments in a local database. When the future SO receives a request to add a comment, it stores the comment in the database; when it receives a request for a document, it fetches the document from its delegate and appends the comments to the document.

The transform function for this upgrade simply merges the name-document and name-comment mappings into a single table that is used by the new version of DocServer. The past SO has no persistent state of it own, so the TF does not need to do anything for it.

When the past SO receives a request for a document, it requests the document from its delegate and removes the comments. Since the delegate just returns a single string, we had to introduce a delimiter between the document and the comments so that the past SO could remove the latter. If such a delimiter were not provided, the past SO would have to disallow document requests.

The future SO implementation is 93 lines of C++, and the past SO is 60 lines. In Python, the same two SOs are 32 lines and 12 lines, respectively. About half of each C++ SO implementation is boilerplate code that could be generated automatically. One thing that greatly simplified implementing the SOs was the use of a database with a simple map interface.

8.2.2 Transform Functions

To evaluate the difficulty of writing transform functions, we created a TF that adds an access control list to every file and directory in a file system. This is done as described in Section 2.5, except access rights are expressed in the AFS [61] "`rlidwka`" format, and the TF also creates ACLs for directories. The ACL format is that of SFSACL [64]: the first 512 bytes of a file's contents contain its ACL, which is a block of text that starts with `ACLBEGIN` and ends with `ACLEND`. Each line in between defines the permissions for a user or group. The ACL for each directory is kept in a file called `.SFSACL` in that directory.

The TF traverses the file system, adding ACLs to files and directories along the way. The initial contents of a file's ACL are derived from the Unix permissions of that file. Owner permissions are mapped to an ACL entry that grants the same permissions to that user; group permissions are mapped to an ACL entry that grants the same permissions to that group; and world permissions are mapped to an ACL entry that grants the same permissions to the special user `sys:anonymous`. This TF assumes that no additional ACL state is kept by the future SO (as described in Section 3.5); supporting this extension is straightforward.

Adding ACLs to directories is easy: the TF just creates the appropriate `.SFSACL` file. But adding ACLs to files is more difficult, since the ACL must be inserted at the beginning of each file. To do this, the TF copies the file to a temporary location, writes the file's ACL to the file's original location, then appends the the file's contents to the ACL. Thus, the execution time of the TF is dominated by the time required to copy each file's contents twice.

We implemented this TF as a 162-line Python script. The implementation was straightforward: it uses the `os.walk` library function to traverse the file system, then transforms each directory and file as it is encountered. In Section 10.3.1, we discuss how this kind of transform can be done *incrementally* to reduce node downtime.

8.2.3 Scheduling Functions

To see how an upgrade schedule works in a large system, we upgraded a DHash system deployed on PlanetLab [87] using 205 physical nodes with 3 virtual nodes each, for a total of 615 virtual nodes. The is the null upgrade (it makes no changes to the server software), so no SOs are needed. The scheduling function upgrades nodes gradually: it flips a biased coin periodically and signals if the coin is heads; we used a heads probability of 0.1 and a period of 3 minutes between flips (this SF

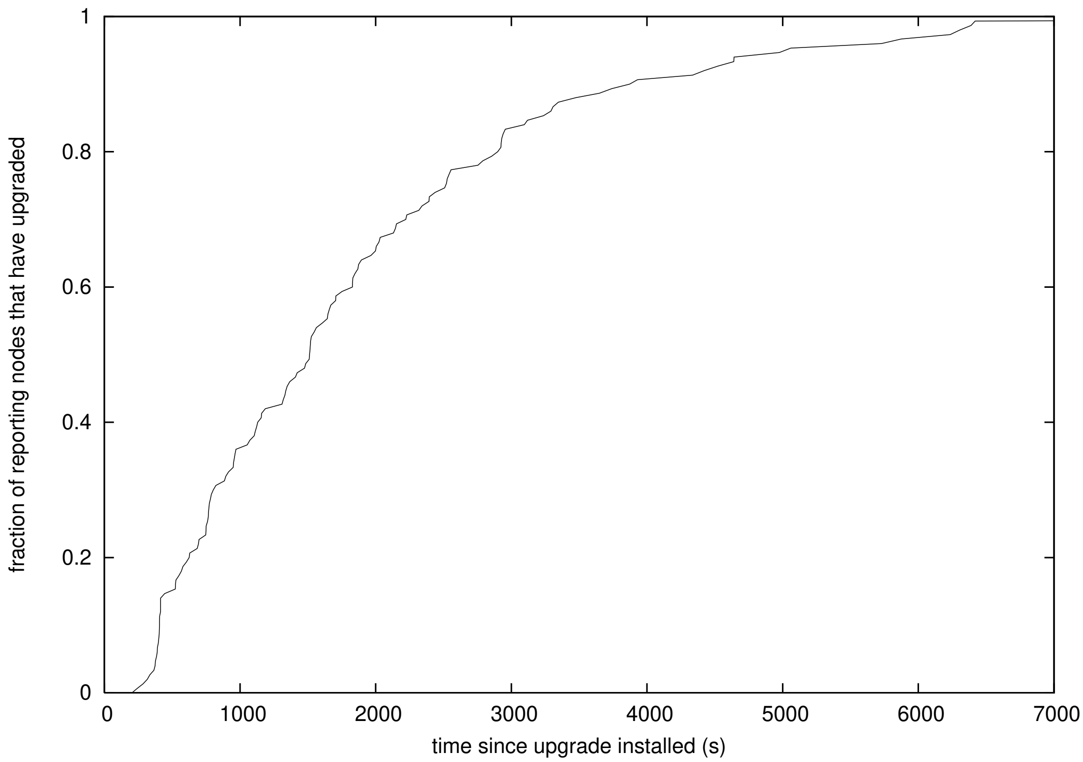


Figure 8-7: *Cumulative fraction of upgraded nodes on PlanetLab. Fraction is out of 151 reporting nodes; excludes 54 nodes that fail to report due to message loss or node failure.*

is implemented as a 6-line Perl script). We set the time limit for the scheduling function to 6000 seconds (100 minutes); by this time, we expect 97% of nodes to have upgraded.[1]

Figure 8-7 depicts the progress of node upgrades as reported to the UDB. The fraction of upgraded nodes increases smoothly up to 6000 seconds, when it jumps up to nearly 100% because of the SF time limit. One node (not shown) does not report having upgraded until 16000 seconds (4.5 hours) have passed, which means either its report to the UDB was delayed that long, or the node downloaded the upgrade more than 10000 seconds after the upgrade was installed! (Our trace for this experiment does not indicate which case occurred.) Several other nodes never report at all, which means either their software failed or their reports were lost in the network (we have since implemented a periodic retransmission to fix the latter problem). Finally, while we intended to have a client trace for this upgrade, the DHash client froze shortly after the upgrade began and never recovered. We conclude that either DHash is not as robust to failures as we might hope, or the scheduling function we chose was too aggressive. Further experiments are warranted to determine the cause of this failure.

[1]The probability that a node has upgraded after n seconds is $1 - ((1 - p)^{n/s})$, where p is the heads probability (0.1) and s is the seconds between flips (180).

From this experience, we learned much about the perils of running distributed systems on the Internet; and while our prototype works pretty well, it is far from perfect. Nonetheless, our design made it easy to define these upgrades, and we believe further work on the implementation can make it robust to the vagaries of large-scale distributed systems.

Chapter 9

Related Work

This chapter reviews research and real-world techniques for providing automatic software upgrades for distributed systems. Our annotated bibliography provides additional details on several of the systems we discuss [20].

What sets our approach apart from all previous approaches is that ours is *realistic* and *comprehensive*. Our upgrade system works in environments where failures are common, whereas previous approaches stall when failures occur [21, 24, 27–29, 59, 66, 90]. Our upgrade model is modular and does not restrict how people implement their systems, whereas previous approaches require a particular object system [21, 24, 27–29, 59, 66, 90, 102]. Our methodology explains how to define the relationship between the types affected by an upgrade and implement these types using simulation objects, whereas previous work falls well short of the level of expressive power we provide [81, 93, 97, 102]. We introduce a new way to define upgrade schedules (using scheduling functions) that, to our knowledge, appears in no previous work. Finally, we describe an infrastructure that is generic and a prototype implementation that provides practical performance for real Internet systems, unlike many previous approaches that are limited to research systems.

We begin, in Section 9.1, with a discussion of systems that support *mixed mode*, i.e., systems that allow objects running different versions to interoperate. These are the approaches most closely related to ours, as they enable systems to provide service while objects upgrade asynchronously. However, none of these approaches provides as much expressive power as ours.

In Section 9.2, we discuss approaches that support a limited form of mixed mode using *extensible* protocols. Such protocols are common in real-world systems, since they enable objects to upgrade asynchronously yet do not (necessarily) require a special infrastructure. Instead, the burden of ensuring interoperability is on the protocol designer.

Then, in Section 9.3, we consider approaches that avoid mixed mode altogether. Most real-world and research approaches to upgrading distributed systems fall into this category. All are far more limited than our approach.

Finally, in Section 9.4, we discuss various approaches for ensuring the state of a system survives upgrades. Our approach is unusual in that it preserves only persistent state. Several approaches preserve volatile state, and some are able to do this without restarting the node. We also discuss how memory-only systems can support extremely rapid upgrades without losing state.

9.1 Supporting Mixed Mode

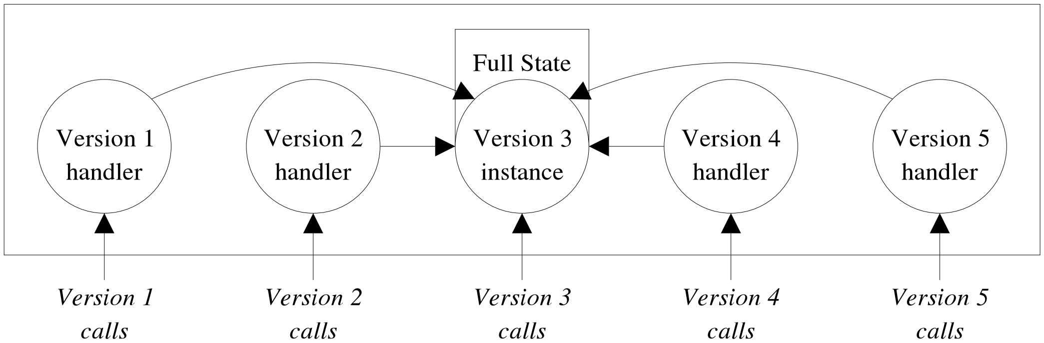
In Section 4.1, we discussed several models for supporting mixed mode in distributed systems. We review these here and provide more detail on the approaches closely related to ours.

PODUS [48] supports upgrades to individual procedures in a (possibly distributed) program. Procedures can be upgraded asynchronously, and user-provided *interprocedures* translate calls intended for the old version of a procedure into calls for the new version. Unlike simulation objects, interprocedures are stateless and cannot chain together to support multiple versions (Figure 9-1(a)).

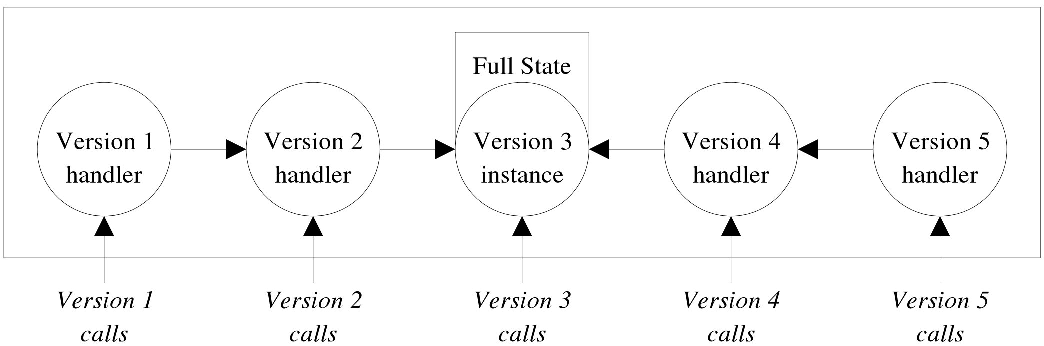
The Eternal system [102] supports rolling upgrades for replicated CORBA objects. Replicas of different versions can interoperate using *wrappers* that translate calls between them. Like PODUS's interprocedures, Eternal's wrappers are stateless and cannot chain together (Figure 9-1(a)). Furthermore, this work pays little attention to the semantics of wrappers, and simply recognizes that they can be useful to support asynchronous upgrades.

The work most closely related to ours is Senivongse's "evolution transparency" model for distributed services [93]. Senivongse describes how to use *mapping operators* to enable cross-version interoperation during distributed upgrades. Senivongse proposes using chains of mapping operators to support multiple versions (Figure 9-1(b)), using backwards mapping operators to enable non-upgraded nodes to support new versions before they upgrade (like future SOs), and deprecating old mapping operators when they are no longer needed (like retiring past SOs).

Senivongse's semantic model for nodes that support multiple types is "evolution transparency:" clients should not notice when they are using an object of a version different than they expect. This is much weaker than what we propose, as Senivongse's model does not capture the relationship



(a) Unchained Handlers



(b) Chained Handlers

Figure 9-1: *Unchained handlers vs. chained handlers. Each node (large box) supports calls at versions 1 through 5. Arrows indicate the direction of method calls. The node runs an instance of version 3's type and has handlers for the other versions.*

between different types. In contrast, our model provides invariants between the states accessible via different types and guarantees on how calls to one type are reflected on the others. This lets clients know what they can expect when they upgrade and change which type they use or when clients running different versions communicate about the state of a node.

9.1.1 Schema Evolution

Many of the same issues that arise in upgrading distributed systems also arise in schema evolution [31, 32, 81, 97] for object-oriented databases (OODBs). For example, a computation in an OODB may require that one object call the methods of another, even though one of the objects has upgraded to a new schema, but the other has not. Some approaches transform the non-upgraded object just in time for the method call [31]. But others [81, 97] use mixed mode: they allow objects of different versions to interact by enabling objects to implement multiple types.

The Encore system [97] wraps each object with a version set interface (VSI) that can accept any method call for any version of the object's type. If the object cannot satisfy a call directly, handlers in the VSI can substitute a response. Unlike simulation objects, these handlers are stateless and cannot chain together (Figure 9-1(a)). As a result, the number of handlers grows quadratically in the number of versions: each time a new version is defined, new handlers must be defined for each old version.

Monk and Sommerville's system [81] uses "update" and "backdate" functions to convert calls from any version to the version of the instance. Like simulation objects, these functions can chain together, so each new version requires only two new functions (Figure 9-1(b)). But unlike simulation objects, these functions are stateless and so cannot implement stateful behaviors.

Some schema evolution systems [32] use *views* [108] to enable objects of different versions to interact. Views are related to mixed mode systems in that they enable a single set of base objects to be accessed via multiple schema.

O2's view system [22] provides a comprehensive study of how mutations made to one type (a view type) are reflected on another (the base type) and so has much in common with our model for supporting multiple types on a single node. In O2, mutations made to view objects are reflected on the base objects via a *translation* layer. The database checks that the new values for the base objects and view objects obey the view definition, and if not, rejects the mutation. This is similar to how a simulation object must disallow calls whose effects cannot be reflected correctly on its delegate. But an important difference is that the SO implementor must determine which calls to disallow, whereas O2 enforces adherence to the view definition at runtime. This is possible because view definitions are given in a form the database can understand, and the database can roll back the effects of a transaction that contains a rejected mutation.

O2 allows views to stack (i.e., the base of a view may be another view), much as we allow simulation objects to chain. O2 furthermore supports a tree of views (i.e., different views may share the same base). Our system restricts nodes to a linear chain of SOs so that upgrades can change which type is implemented by the current object and which types are implemented by SOs.

9.1.2 Related Approaches

Federated distributed systems [45, 78, 88, 94] address similar challenges as those posed by mixed mode systems. Federated systems may need to transform messages as they pass from one domain of the system to another, e.g., a system may transform a monetary value from one currency to another as it is transferred between different countries. Unlike our work, these approaches provide little information on the semantics of transformation.

Richardson [89] describes how to integrate wrappers (called *aspects*) into a type system, so that a single object may evolve over time with new state and new behavior. Like simulation objects, aspects are stateful. But unlike simulation objects, aspects are not constrained to maintain any relationship between their state and that of the underlying object. In particular, calls made via one aspect may change the state of the underlying object without the knowledge of the other aspects, so the states of those other aspects may become out-of-sync with that of the underlying object.

9.2 Limited Mixed Mode

Many systems, such as Google [34], Gnutella [15], Internet email [41], and the World Wide Web [82], support a limited form of mixed mode via extensible protocols. Extensible protocols can be thought of as having two parts: a baseline protocol and a set of extensions. All nodes understand the baseline protocol, and the system provides "acceptable service" using just this protocol. For example, all mail agents support the `To`, `From`, and `Subject` headers, and these are enough to provide basic email service.

Extensions are enhancements to a protocol that some subset of the nodes in a system understand, for example, the `X-Spam-Level` email header. Nodes ignore extensions they don't recognize, though they may forward messages that contain extensions to other nodes that do recognize them. This means system administrators can introduce new behavior via extensions without disrupting baseline service.

The problem with extensible protocols is that nodes must function correctly with or without the extensions they recognize. This complicates software design, and an administrator can never rely on all nodes in the system supporting a given extension. The only way to ensure that all nodes support a new behavior is to change the baseline protocol, and this requires a real upgrade.

Some research systems [53, 83] use extensions to support upgrades to objects in distributed systems. Govindan [53] proposes an active distributed service (ADS) composed of cooperating agents that can be extended by plugging in new message handlers. The Information Bus [83] allows objects to communicate using publish-subscribe, so objects of different versions can interact by subscribing to just those messages that they understand. Others [38, 106] have also advocated the use of publish-subscribe to loosen component coupling and make reconfiguration easier.

9.2.1 Compatible Upgrades

Another way systems support limited forms of mixed mode is by requiring that new software versions be "compatible" with old versions. Compatibility means clients of the old version can use the new one transparently. This a limited form of mixed mode because it does not allow upgraded nodes to call methods of non-upgraded nodes. Therefore, while it can be used for client-server or multi-tiered systems, this approach will not work for server-to-server or peer-to-peer systems.

Compatibility does not simply mean that the new type has all the methods of the old type. Bloom [29,30] defines "legal" (compatible) replacements as those that "preserve or invisibly extend the continuation abstractions" of the original versions. In our work, we define compatibility to mean the new type of an object is a behavioral subtype [73] of the old type.

Several upgrade systems [40,74,95] leverage compatibility not to support mixed-mode but rather to test (at runtime) whether a new object provides the same behavior as the old one. These systems run the two objects side-by-side, dispatching calls to both. Clients continue to use the old object until the system (or an administrator) decides that the new object correctly implements the old behavior and causes the new object to take over.

McCamant [77] describes how to predict problems caused by component upgrades before an upgrade is installed by combining dynamic invariant detection with a system's test suites.

9.3 Avoiding Mixed Mode

Supporting mixed mode adds complexity to a system, and complexity threatens a system's reliability. Therefore, many systems strive to avoid mixed mode by allowing service to degrade during upgrades.

System administrators often synchronize the software on a set of nodes using utilities like `rsync` [104] and package installers [1, 10, 18]. The system does not provide service while this is happening, so administrators typically run these utilities when users are inactive.

Centrally-administered software management systems [9,26,55,96] provide better management and monitoring of upgrade progress than the utilities mentioned above, but they do nothing special to enable nodes running different versions to interact. Thus, service may degrade while an upgrade is in progress.

Many systems use custom techniques for upgrading their nodes [33]. Google [51] upgrades entire data centers at once and provides service by redirecting clients via DNS to other, redundant data centers (that are not upgrading). Since all the nodes in a data center upgrade together, nodes running different versions never communicate. This approach supports protocol changes within data centers but not between data centers.

CoDeeN [85] is a content distribution network deployed on PlanetLab [87] that upgrades its nodes about twice a week, causing only about 20 seconds of downtime per node. New versions are simply copied to the nodes, and the software is restarted to use the new code. New versions are typically compatible; on the rare occasions when a new version is incompatible, version numbers are used to distinguish new calls from old ones (which are rejected).

Gnucleus [7] is a peer-to-peer file-sharing system that disseminates upgrades eagerly by gossip. Nodes upgrade as soon as they hear about new versions, so Gnucleus does not need to support mixed mode. However, service may degrade (e.g., searches for files may fail) while the upgrade is in progress.

Online games [2, 11] force clients to upgrade to the latest version before they can participate in games. This results in delayed service to clients, but it allows the system to avoid supporting mixed mode. Service is also interrupted when servers upgrade.

Reconfigurable distributed systems enable the replacement of object implementations in distributed object systems, such as Argus [29], Conic [66], Durra [24], Polylith [59], Olan [27], CORBA [21,28], C2 [84], and Java [90]. These approaches allow whole subsystems (collections of objects) to upgrade together, but the new type provided by a subsystem must be compatible with the old one. Some approaches [21,24,27,28,59,66,84] isolate the parts of the system that are upgrading by "quiescing" the upgrading nodes and/or the links between those nodes and the rest of the system; therefore, service degrades when large parts of the system upgrade. JDrums [90] alleviates this problem by upgrading objects lazily and keeping old versions around so that old references continue to work. Argus [29] and the Eternal system [102] use distributed transactions and totally-ordered multicast, respectively, to serialize reconfiguration in the computation of a system, thus preventing nodes of different versions from interacting.

IBM's K42 operating system uses dynamic interposition to support hot-swapping (runtime replacement) of OS components [98]. In the common case, nothing is interposed between callers and OS components; interposers are installed dynamically by modifying a call indirection table. Interposition enables hot-swapping using the Read-Copy Update (RCU) scheme: an interposed Mediator blocks new calls to the component, lets the old calls drain, transfers state to the new component, then unblocks the calls. This scheme assumes that requests to a component are short-lived, so that it is reasonable to block new calls while old calls drain and while the component is replaced.

9.4 State Management

This section discusses how various approaches guarantee that the state of a system is preserved across upgrades.

Many upgrade systems [29, 46, 48, 54, 57, 59, 102] support the transfer of state between versions via *abstract value transmission*, i.e., they require that the pre-upgrade object export its state as an abstract data value and the post-upgrade object import this value to initialize its state. The correctness of this approach derives from Herlihy and Liskov's value transmission technique for abstract data types [56].

The problem with the value-transmission approach is it requires that software implementors create the state export and import routines. Some approaches alleviate this problem by providing tools that automatically generate these routines [57, 68, 102].

Our approach takes advantage of the fact that robust systems typically preserve their state in persistent form so that the state survives node restarts. Therefore, an upgrade can transfer state between versions simply by changing the persistent state of a node from the representation required by the old object to that required by the new object. This approach requires no additional routines beyond the recovery routines that the implementor must already provide. Furthermore, this approach is more likely to capture the complete state of a node than value transmission, because persistent state is used regularly for recovery, whereas import/export may miss some state.

The problem with transforming persistent state is that it is slow. One way to avoid this problem is to avoid using persistent state altogether. Systems like Google [34] and SSM [70] keep their entire state in memory and provide reliability using replication. In these systems, a node can restore its state from a replica in just a few seconds over a gigabit ethernet. By restarting nodes in rapid succession, these systems can upgrade with very little downtime.

However, memory-only systems are expensive if the state is very large: they require thousands of nodes (each with a large memory), a high-speed network, and a long-lived backup power supply. To survive catastrophic failures, these systems require multiple data centers, and these data centers may need to upgrade independently (e.g., for testing) [51].

Dynamic software updating [46, 48, 52, 54, 57, 58, 75] enables nodes to upgrade their code and transform their volatile state without shutting down. These techniques are typically language-specific and require that the implementor provide extra information to enable programs to be updated, such as "reconfiguration points" that identify where in the program reconfiguration can take place. We chose a different approach because upgrades are rare and failures are common: it is okay to discard volatile state when a node upgrades, therefore we can avoid the complexities of dynamic updating. Nonetheless, these approaches are complementary to ours and could be used to reduce downtime during upgrades.

Chapter 10

Conclusions

Long-lived distributed systems must evolve with changing technologies and requirements. But system evolution must not disrupt service, as users increasingly rely upon these systems as critical resources. Existing approaches to upgrading distributed systems are unsatisfactory: they disrupt service during upgrades, restrict how systems are implemented, and restrict how software may evolve.

This thesis describes a new *automatic software upgrade system* that supports unrestricted software changes for distributed systems and enables those systems to provide service during upgrades. We make two major contributions: a methodology for defining upgrades and an infrastructure for deploying upgrades automatically. Together, the methodology and infrastructure provide a complete, practical solution to the problem of upgrading long-lived distributed systems.

10.1 Methodology

Our approach allows nodes in a system to upgrade *asynchronously*, i.e., independently and at any time. This enables upgrades to be *scheduled* so that the system can provide service while an upgrade is in progress. Our approach leverages the fact that long-lived systems are *robust*: they tolerate node failures and allow nodes to recover and rejoin the system, and furthermore, nodes keep their important state on persistent storage so that it survives failures. This means the upgrade system can restart nodes that need to upgrade and discard their volatile state.

The key challenge for our approach is that there may be long periods of time when a system is in *mixed mode*, i.e., when different nodes are running different software versions and yet need to communicate. To address this challenge, our approach enables nodes to implement multiple types simultaneously: one for each version that clients might expect. This allows each node to interoperate with other nodes as though all the nodes in the system were running the same version, while in reality nodes may be separated by several versions. Our approach is *modular*: defining an upgrade only requires understanding the new and current versions of the system software.

Enabling a node to implement multiple types requires a new way to specify the relationship between two types; this is described in Chapter 3. This relationship defines how different clients interacting with the same node via different types see the effects of one another's actions. This also allows clients to know what node state they can expect to see when they upgrade and start using a node via a new type.

The effects of calls to one of a node's types must be reflected on all its types, but sometimes this isn't possible because of incompatibilities. In such cases, a node must *disallow* calls that would violate these requirements, i.e., the node must cause those calls to fail. Clients are prepared for such failures and work around them when possible. Thus, systems can degrade service gracefully when upgrades are incompatible, rather than halt.

A node implements multiple types using *simulation objects*, which are discussed in Chapter 4. A simulation object is an adapter that implements one type by delegating to another. Simulation objects can have their own persistent state and so can implement more powerful behaviors than the adapters proposed in previous work [48, 81, 93, 97, 102]. Different designs for using simulation objects offer different tradeoffs in terms of the behaviors they can simulate, so choosing the right design depends on how a system is likely to be upgraded.

An upgrade must preserve the state of a system from one version to the next; Chapter 5 explains how this is accomplished using *transform functions*. A transform function reorganizes the persistent state of a node from the representation used by the old software to that used by the new software. This transformation is transparent to clients: they see the same state after the upgrade as they saw before, regardless of whether they were using the old type or the new one.

How an upgrade is scheduled depends on a variety of factors, including a system's physical organization and the relationship between the old and new software versions. *Scheduling functions* provide a powerful way to implement a variety of upgrade schedules, as discussed in Chapter 6. Our system provides several sources of information to aid in scheduling and monitoring upgrades, including a central *upgrade database* that stores information about each node's upgrade status.

10.2 Infrastructure

The components of an upgrade—simulation objects, transform functions, and scheduling functions—describe how nodes move from one version to the next and support multiple types. But this is only half the solution; actually deploying upgrades and enabling a system to support mixed mode requires an *upgrade infrastructure*.

The upgrade infrastructure is a platform for distributed systems that deploys upgrades automatically and handles cross-version communication. A central *upgrade server* stores upgrade definitions and makes them available for download. Per-node *upgrade layers* download and install upgrades and handle cross-version messages by dispatching them to the appropriate simulation objects. The upgrade infrastructure is transparent to the system it supports and requires no special changes to the system software.

Chapter 7 describes Upstart, our prototype implementation of the upgrade infrastructure. Upstart is generic: it works with any system that uses sockets to communicate. Our implementation includes libraries that make working with Sun RPC [99] applications particularly convenient, but Upstart works just as well with other protocols and even raw TCP streams.

Upstart is practical: its overhead is reasonable for all but the most demanding applications, as shown in Chapter 8. We have evaluated Upstart with small test applications and large, complex peer-to-peer systems on high-speed local networks and on the Internet. Upstart has no inherent scalability limitations, as load on the central upgrade server is alleviated using a software distribution network.

10.3 Future Work

There are several directions for extending this work. One of the most obvious is to improve Upstart's usability. Currently, deploying an upgrade involves editing configuration files and running several scripts; this process could be made much more natural with better tools.

The most interesting areas for exploration involve extending our approach to be more efficient, more robust, and more general. In the following sections, we discuss several such areas.

10.3.1 Incremental Transform Functions

A transform function may take a long time to execute if a node's state is large or if the transform is complex. This time can be reduced using an *incremental* transform function. An incremental TF transforms pieces of the state on demand, i.e., when they are accessed by the node's object. This has two benefits: the node spends very little time unavailable (the TF just needs to mark the state as "untransformed"), and the node only spends time transforming state that it uses.

An example of an incremental TF is changing the representation of files in a file system (e.g., adding an access control list to each file, as described in Section 8.2.2). The TF only transforms a file when the new class (e.g., the file server) attempts to access that file.

Incremental TFs have disadvantages: they are more complex than normal ("eager") TFs, and they introduce runtime overhead when the object accesses its state. Furthermore, transforms that must read the whole old state (in the old representation) to produce the new state cannot be incremental.

Incremental TFs make sense for state that has many independent pieces (e.g., files in a file system, records in a database, or objects in an OODB) that can each be transformed quickly. A possible approach to implementing incremental TFs is to leverage existing support for automatic transformation in persistent storage systems, e.g., lazy schema evolution in OODBs [31]. Thus, the transform function for a node upgrade can be implemented as a schema change for the storage system used by the node.

10.3.2 Dealing with Errors

The most frightening thing about upgrading a system is the possibility of introducing errors [105]. Clearly, such errors may come from the new implementation installed by the upgrade. But an automatic upgrade system offers two more sources of errors: the upgrade definition (its scheduling function, simulation objects, and transform function) and the upgrade infrastructure itself. In this section, we present ideas for how to deal with these three kinds of errors.

Errors in the new implementation

A simple way to fix a buggy implementation is to run a subsequent upgrade that replaces the buggy implementation with a good one. But this takes too long, because the upgrader has to fix the bugs and define the new upgrade, and all the while the system is providing buggy service.

People in industry seem to agree that automatic upgrade systems should provide the ability to undo or "roll back" an upgrade when an error is detected [33, 44, 105]. Many systems support upgrade undo using *staging* [33, 40, 95]. The idea is that the new implementation runs in a special staging area on each node until the administrators determine whether it is working correctly. If the new implementation is okay, then the nodes move it from the staging area to the "real" area. Otherwise, the nodes revert to the old implementation.

Staging is simple to implement and makes it easy to revert back to an old implementation, but it requires that nodes have enough storage space for both the old and new implementations and their associated state. It is important for the old and new implementations to each have their own state, because the new implementation may need to transform its state to a new representation and, if the new implementation is buggy, it may corrupt its state.

One way to avoid the extra overhead of keeping two copies of the node state is to instead keep a log of state changes made by the new implementation. If it turns out the new implementation is buggy, the node can use the log to undo the changes to the state.

In either case, reverting to the old version causes a node to lose the effects of all operations that occurred after it started running the new (buggy) implementation. Given a log of those operations, it may be possible to "replay" them on the old (good) implementation and restore their effects [35]. If old implementation does not understand the new operations, replaying may require a simulation object to convert the new operations to old ones.

Errors in the upgrade definition

Our approach to upgrading distributed systems requires that the upgrader provide several pieces of code: scheduling functions, simulation objects, and transform functions. If these have bugs, then the upgrade may not work, may disrupt service, and may even corrupt state.

Buggy scheduling functions are not too serious, since timeouts ensure all nodes eventually upgrade. Buggy simulation objects and transform functions are more serious problems, as they may destroy or corrupt the state of a node (while SOs cannot corrupt the state of the current object directly, they can still mutate its state by calling its methods). Staging can alleviate this problem, as it makes it possible to undo a buggy upgrade and restore the uncorrupted state. If nodes do not have enough room to store both the old and new state, we can instead upgrade just a few nodes to test the transform and keep backups of their state on replicas.

To reduce the occurrence of bugs in upgrade definitions, the upgrade system should provide tools for testing upgrades and checking the correctness of the upgrade components. Simulation objects and transform functions have well-defined specifications, so developing techniques for checking that they meet these specifications is an interesting area for research. (Of course, the upgrade specification itself may also have bugs!) McCamant [77]'s techniques for predicting problems caused by component upgrades might be useful to help determine whether an SO satisfies its specification.

Scheduling functions do not have well-defined specifications. However, one can often express an upgrade schedule as a set of constraints, e.g., "upgrade servers before clients," "upgrade server replicas at different times," and "upgrade clients at night." It seems possible to use such constraints to check or even generate the scheduling functions for an upgrade.

This leads us to the question of whether we can also generate the other upgrade components. Previous upgrade systems provide support for generating transform functions [57, 68, 102] (but not for persistent state) and wrapper functions [93, 102] (which are similar to simulation objects, but less sophisticated). But most of these approaches just generate skeleton code, and none provide assurances on the correctness of the generated components. Clearly, further research is possible here.

Errors in the upgrade infrastructure

Problems with the upgrade server, software distribution network, or upgrade database can be dangerous, but most such errors are likely to delay upgrades rather than cause any real damage. However, problems with the upgrade layer can be disastrous, as they could disrupt communication throughout the system and corrupt the software and state of nodes.

Repairing such problems requires replacing the upgrade layer on every node in the system. This is like an upgrade, except we cannot use the upgrade layer to do it! Instead, we need a simple and reliable daemon process on each node that enables us to install new upgrade layer software on every node. This daemon does not intercept node communication, so there can be no simulation objects for this upgrade. This means we need to minimize the period during which nodes are running different versions of the upgrade layer; we do so by running this upgrade eagerly.

Each node's daemon periodically polls the upgrade server to determine whether a new version of the upgrade layer software is available. The upgrade layer version (UL version) is distinct from the system software version, and upgrade layers include their UL version in the headers that they exchange periodically. When an upgrade layer receives a header whose UL version differs from its own, it notifies its daemon of the new UL version (this is just an optimization, since the daemon also polls the upgrade server).

When the daemon learns of a new UL version, it downloads the new UL software (and an optional UL transform function) from the software distribution network, shuts down the node, installs the new UL software, runs the TF, and restarts the node. This process is quick and sweeps rapidly across the system; nonetheless, it will disrupt service.

If the upgrade to the upgrade layer is not urgent, we will want a less disruptive way of deploying it. We can accomplish this by allowing UL upgrades to include scheduling functions and by enabling ULs of different versions to communicate. It does not seem necessary to use something as sophisticated as simulation objects for this; instead, we could use something simpler but less powerful, like the extensible protocols discussed in Section 9.2.

10.3.3 Extending the Model

There are several ways to extend our upgrade model to support more kinds of upgrades and systems.

Filters

Our discussion has assumed that a class upgrade replaces all instances of an old class with instances of a new class. In Section 2.2, we proposed that an upgrader could restrict a class upgrade to only some nodes belonging to the old class using a *filter*. In this section, we discuss some of the issues involved in supporting filters.

Filters complicate how a node determines whether it is affected by an upgrade. Without filters, a node just checks whether its current class matches the old class of any class upgrade in the next upgrade. There will be at most one class upgrade whose old class matches, so there is no ambiguity.

With filters, there may be multiple class upgrades that affect the same old class. A node determines which class upgrade affects it by checking whether the filter for each class upgrade in turn "matches" the node. The node checks the filters in the order they are presented in the configuration file and installs the upgrade whose filter matches first. If no filter matches, the node is unaffected by the upgrade.

So what is a filter? It could be just a boolean expression in some language that the nodes understand. For example, a class upgrade could have the filter `bandwidthTo(12.34.56.78) > 10 Mbps`, meaning the upgrade applies only to nodes whose bandwidth to the host with IP address 12.34.56.78 is greater than ten megabits per second. But choosing this language is tricky, and this language itself may need to evolve over time.

Instead, we can implement filters as downloadable routines, like scheduling functions. When a node learns of a new version, it downloads a *filter function* defined for its current class in the system configuration. The filter function returns an identifier for the new class of the class upgrade that the node should next install, or "none" if the node is unaffected by the new version.

This model avoids the need to introduce a new filter language and allows the filter function to examine the node state directly (perhaps using a meta-observer, as discussed in Section 6.2.2). But the node cannot run a filter function before it has upgraded to the class for which the function is defined, because otherwise it may not be able to implement the observers called by the filter.

The filter function must be restricted so that it is certain to terminate. In particular, it must not loop forever or wait indefinitely on remote method calls. One way to ensure that the filter function terminates with a valid return value is to define a time limit and a default return value for it, but this only works when a sensible default exists.

Message-Passing Systems

Our approach models a distributed system as a set of objects that communicate using remote method calls. This is appropriate for systems that use RPC [99] or RMI [79], but not for systems that use general (one-way) message passing, like sensor networks. We would like to be able to reason about upgrades for message-passing systems in the same way we do about RPC systems.

Many uses of one-way messages can be modeled as RPCs. For example, heartbeat messages can be modeled as RPCs that have no return value or exceptions. Recursive lookups (as in DNS) can be modeled as RPCs whose return value comes from a different node than the one that received the call. Cumulative acknowledgments (as in TCP) can be modeled as return values for sets of outstanding RPCs. It may be possible to use such models to reason about upgrades and simulation in message-passing systems.

Implementing upgrades for message-passing systems is straightforward, as Upstart already allows simulation objects to manipulate individual messages. However, Upstart is too heavyweight for systems like sensor networks. Developing a lightweight upgrade infrastructure for sensor networks is an interesting area for further research; the Trickle [69] software dissemination protocol may be a useful building block for such a system.

Security

An important issue in automatic software management is security [43]. Our infrastructure protects the integrity of the system configuration, software, and upgrade definitions using digital signatures; but we ignore issues of authenticating the nodes in the system (i.e., ensuring upgrades are provided only to licensed users) and protecting the privacy of software components (because of their intellectual property value). Providing these features requires straightforward extensions to our infrastructure.

10.4 Conclusion

This thesis has presented a comprehensive approach for providing automatic software upgrades for long-lived distributed systems. The approach is realistic: it works in environments where failures are common and it is impractical to upgrade whole systems at once. The approach is modular and defines precise rules for reasoning about upgrades. The approach includes an infrastructure that scales and performs well. Nonetheless, there remain many issues to explore, and I hope this work inspires further research.

Appendix A

Configurations

The *configuration* of a system resides on the upgrade server and defines the schema for each version of the system. A configuration has an minimum version number (identifies the minimum active version), one or more initializers, and zero or more subsequent versions:

$$config \;\; = \;\; \langle number, initial+, \; version* \rangle$$

- $number \geq 1$

The initializers identify the classes that belong to the version 1 schema and define their implementations. Note that the minimum active version may be greater than 1.

$$initial \;\; = \;\; \langle classID, \; code, \; library \rangle$$

- *code* implements class *classID*

- *library* defines *createProxy*

The library in each initializer defines a static factory that creates proxy objects for that class. Proxies are described in more detail in Chapter 7.

Versions define one or more class upgrades. Each class upgrade specifies an oldclass whose instances will be replaced by instances of newclass.

$$\begin{aligned} version \;\; &= \;\; \langle number, \; upgrade+ \rangle \\ upgrade \;\; &= \;\; \langle oldclassID, \; newclassID, \; library, \; type, \; code, \; sf, \; sfMaxSecs, \; tf \rangle \end{aligned}$$

$$library \quad = \quad \langle createProxy,\ createPastSO?,\ createFutureSO?,\ createBridge? \rangle$$

- *code* implements class *newclassID*
- Let $config = \langle num, inits, [v, \ldots] \rangle$, then
  $v.number = num + 1$ and $oldclasses(v) = inits.classID$
- Let $config = \langle num, inits, [\ldots, u, v, \ldots] \rangle$, then
  $v.number = u.number + 1$ and $oldclasses(v) = oldclasses(u) - u.upgrade.oldclassID$
- all $v : version \mid v.upgrade.oldclassID \subseteq oldclasses(v)$
- $type \in \{$`sametype`, `subtype`, `supertype`, `unrelated`$\}$
- If $type =$ `unrelated`, then *library* defines *createBridge*
- If $type \in \{$`subtype`, `unrelated`$\}$, then *library* defines *createFutureSO*
- If $type \in \{$`supertype`, `unrelated`$\}$, then *library* defines *createPastSO*

The *SF* determines when a node running *oldclass* should upgrade, and the *TF* produces state for *newclass* from the state of *oldclass* and the newclass future SO. The *sfMaxSecs* attribute is a failsafe that causes nodes to upgrade at most *sfMaxSecs* seconds after the SF is invoked.

We can extend the model to allow new classes to be introduced directly in later schema (rather than just as replacements for earlier classes):

$$version \quad = \quad \langle number,\ (upgrade \mid initial)+ \rangle$$

Extending the model in this way (and extending `upstart` and `upcheck` appropriately) is future work.

The state of a *node* is determined by the installation of an initializer and subsequent class upgrades. A node has a current class; current, minimum, and maximum supported version numbers; and libraries that define SOs and proxies for certain versions:

$$\begin{aligned} node \quad &= \quad \langle classID,\ minv,\ curv,\ maxv,\ libraries \rangle \\ libraries \quad &= \quad \langle libv,\ library \rangle + \end{aligned}$$

- $minv \leq curv \leq maxv$

- $\langle curv, l \rangle \in libraries$, and l defines *createProxy*

- If $\langle v, l \rangle \in libraries$, then $minv \leq v \leq maxv$

Appendix B

Dispatch Tables and Delegation Chains

The following is pseudocode for initializing the dispatch table and delegation chains of a node according to the hybrid simulation model described in Section 4.4. A call for version `v` is dispatched to the object `handler[v]`. The methods `createFutureSO`, `createBridge` and `createPastSO` each take a reference to the delegate of the object to create, and the first two also take a flag indicating whether the object is running as an interceptor.

```
# initialize dispatch table to an empty map
handler = {}
isInterceptor = true

# create proxy
delegate = handler[curv] = upgrade[curv].createProxy()

# create future SOs
for v from curv+1 up to maxv:
  switch upgrade[v].type:
    case skipped or sametype:
      handler[v] = delegate
    case subtype:
      handler[v] = upgrade[v].createFutureSO(delegate, isInterceptor)
    case supertype:
      isInterceptor = false
      handler[v] = delegate
    case unrelated:
      bridge = upgrade[v].createBridge(delegate, isInterceptor)
      # bridge takes over calls for its delegate
      for u from curv up to v-1:
        if isInterceptor or (handler[u] == delegate):
          handler[u] = bridge
      isInterceptor = false
      handler[v] = upgrade[v].createFutureSO(bridge, false)
  # if latest object intercepts, reassign previous versions to it
  if isInterceptor:
    for u from curv up to v:
```

```
    handler[u] = handler[v]
  delegate = handler[v]
# end create future SOs

# create past SOs
delegate = handler[curv] # this may no longer be the proxy
# the past SO for v-1 is defined by the version v upgrade
for v from curv down to minv+1:
  switch upgrade[v].type:
    case skipped or sametype or subtype:
      handler[v-1] = delegate
    case supertype or unrelated:
      handler[v-1] = upgrade[v].createPastSO(delegate)
  delegate = handler[v-1]
# end create past SOs
```

Bibliography

[1] APT HOWTO. http://www.debian.org/doc/manuals/apt-howto/.

[2] Battle.net multiplayer online game server. www.battle.net.

[3] Cisco Resource Manager. http://www.cisco.com/warp/public/cc/pd/wr2k/rsmn/.

[4] Common object request broker architecture (CORBA) core specification. http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[5] EMC OnCourse. http://www.emc.com/products/software/oncourse.jsp.

[6] Folding@Home distributed computing. http://www.stanford.edu/group/pandegroup/folding/.

[7] The Gnucleus open-source Gnutella client. http://www.gnucleus.com/Gnucleus/.

[8] Kazaa. http://www.kazaa.com/.

[9] Marímba. http://www.marimba.com/.

[10] Red Hat up2date. http://www.redhat.com/docs/manuals/RHNetwork/ref-guide/up2date.html.

[11] Sony Online Entertainment. http://sonyonline.com/.

[12] Apache HTTP server project, 1995. http://httpd.apache.org/.

[13] The GNU privacy guard, 1999. http://www.gnupg.org/.

[14] Akamai, 2000. http://akamai.com/.

[15] The Gnutella file sharing protocol, 2000. http://rfc-gnutella.sourceforge.net.

[16] Windows 2000 clustering: Performing a rolling upgrade. 2000.

[17] GNU wget, 2001. http://www.gnu.org/software/wget/wget.html.

[18] Managing automatic updating and download technologies in Windows XP. http://www.microsoft.com/WindowsXP/pro/techinfo/administration/manageau%toupdate/default.asp, 2002.

[19] GMail: A Google approach to email, 2004. http://www.gmail.com.

[20] Sameer Ajmani. A review of software upgrade techniques for distributed systems. August 2002.

[21] Joao Paulo A. Almeida, Maarten Wegdam, Marten van Sinderen, and Lambert Nieuwenhuis. Transparent dynamic reconfiguration for CORBA, 2001.

[22] S. Amer-Yahia, P. Breche, and C. Souza. Object views and updates. In *Proc. of Journes Bases de Donnes Avances*, 1996.

[23] Siddhartha Annapureddy, Michael J. Freedman, and David Mazires. http://www.scs.cs.nyu.edu/~reddy/sahakara/sahakara.html, 2004.

[24] M. Barbacci et al. Building fault-tolerant distributed applications with Durra. In Intl. Conf. on Configurable Dist. Systems [63], pages 128–139. Also in [62], pages 83–94.

[25] Donnie Barnes. RPM HOWTO. http://www.rpm.org/RPM-HOWTO/, November 1999.

[26] T. Bartoletti, L. A. Dobbs, and M. Kelley. Secure software distribution system. In *Proc. 20th NIST-NCSC National Information Systems Security Conf.*, pages 191–201, 1997.

[27] Luc Bellissard, Slim Ben Atallah, Fabienne Boyer, and Michel Riveill. Distributed application configuration. In *Intl. Conf. on Dist. Computing Systems*, pages 579–585, 1996.

[28] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *4th Intl. Conf. on Configurable Dist. Systems*, pages 35–42, Annapolis, MD, May 1998.

[29] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, 1983. Also available as MIT LCS Tech. Report 303.

[30] Toby Bloom and Mark Day. Reconfiguration in Argus. In Intl. Conf. on Configurable Dist. Systems [63], pages 176–187. Also in [62], pages 102–108.

[31] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, California, October 2003.

[32] Philippe Breche, Fabrizio Ferrandina, and Martin Kuklok. Simulation of schema change using views. In *Database and Expert Systems Applications*, pages 247–258, 1995.

[33] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, July 2001.

[34] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[35] A. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to dependability. In *10th ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[36] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[37] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th OSDI*, San Diego, USA, October 2000.

[38] M. R. V. Chaudron and F. van de Laar. An upgrade mechanism based on publish/subscribe interaction. In Workshop on Dependable On-line Upgrading of Dist. Systems [107].

[39] Bram Cohen. BitTorrent, 2001. `http://bitconjurer.org/BitTorrent`.

[40] Jonathan E. Cook and Jeffery A. Dage. Highly reliable upgrading of components. In *Intl. Conf. on Software Engineering*, Los Angeles, CA, 1999.

[41] David H. Crocker. Standard for the format of ARPA Internet text messages. RFC 882, University of Delaware, August 1982.

[42] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Symposium on Operating System Principles (SOSP)*, October 2001.

[43] P. Devanbu, M. Gertz, and S. Stubblebine. Security for automated, distributed configuration management. In *ICSE Workshop on Software Engineering over the Internet*, April 1999.

[44] Chryssa Dislis. Improving service availability via low-outage upgrades. In Workshop on Dependable On-line Upgrading of Dist. Systems [107].

[45] Huw Evans and Peter Dickman. DRASTIC: A run-time architecture for evolving, distributed, persistent systems. *Lecture Notes in Computer Science*, 1241:243–??, 1997.

[46] R. S. Fabry. How to design systems in which modules can be changed on the fly. In *Intl. Conf. on Software Engineering*, 1976.

[47] Michael J. Freedman, Eric Freudenthal, and David Mazires. Democratizing content publication with Coral. In *1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, March 2004.

[48] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, pages 111–128, February 1991.

[49] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20(1):1–24, February 2002.

[50] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 4: Structural Patterns: Adapter, pages 139–150. Addison-Wesley, 1995.

[51] Sanjay Ghemawat. Google, Inc., personal communication, 2002.

[52] Stephen Gilmore, Dilsun Kirli, and Chris Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, University of Edinburgh, December 1997.

[53] R. Govindan, C. Alaettino, and D. Estrin. A framework for active distributed services. Technical Report 98-669, ISI-USC, 1998.

[54] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software Practice and Experience*, 23(9):949–964, September 1993.

[55] Richard S. Hall, Dennis Heimbeigner, Andre van der Hoek, and Alexander L. Wolf. An architecture for post-development configuration management in a wide-area network. In *Intl. Conf. on Dist. Computing Systems*, May 1997.

[56] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, 1982.

[57] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–23, 2001.

[58] Gilsi Hjalmtysson and Robert Gray. Dynamic C++ classes—A lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conf.*, pages 65–76, June 1998.

[59] Christine R. Hofmeister and James M. Purtilo. A framework for dynamic reconfiguration of distributed programs. Technical Report CS-TR-3119, University of Maryland, College Park, 1993.

[60] Markus Horstmann and Mary Kirtland. DCOM architecture, July 1997. Microsoft Distributed Component Object Model.

[61] J. H. Howard. An overview of the andrew file system. In *USENIX Conference Proceedings*, pages 213–216, Dallas, TX, 1988.

[62] *IEE Software Engineering Journal, Special Issue on Configurable Dist. Systems*. Number 2 in 8. IEE, March 1993.

[63] *Intl. Workshop on Configurable Dist. Systems*, London, England, March 1992.

[64] Michael Kaminsky, George Savvides, David Mazières, and M. Frans Kaashoek. Decentralized user authentication in a global file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 60–73, Bolton Landing, New York, October 2003.

[65] Deepak Kapur. Towards a theory for abstract data types. Technical Report MIT-LCS-TR-237, MIT, June 1980.

[66] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[67] B. W. Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, volume 17, pages 33–48, 1983.

[68] Barbara Staudt Lerner. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1):83–127, 2000.

[69] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[70] Benjamin C. Ling, Emre Kiciman, and Armando Fox. Session state: Beyond soft state. In *Networked Systems Design and Implementation (NSDI)*, pages 295–308, March 2004.

[71] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing persistent objects in distributed systems. In *European Conf. on Object-Oriented Programming*, June 1999.

[72] Barbara Liskov and John Guttag. *Program Development in Java*. Addison-Wesley, 2001.

[73] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[74] Chang Liu and Debra J. Richardson. Using RAIC for dependable on-line upgrading of distributed systems. In Workshop on Dependable On-line Upgrading of Dist. Systems [107].

[75] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *European Conf. on Object-Oriented Programming*, 2000.

[76] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 124–139, Kiawah Island, South Carolina, December 1999.

[77] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 287–296, Helsinki, Finland, September 2003.

[78] B. Meyer, S. Zlatintsis, and C. Popien. Enabling interworking between heterogeneous distributed platforms. In *IFIP/IEEE Intl. Conf. on Dist. Platforms (ICDP)*, pages 329–341. Chapman & Hall, 1996.

[79] Sun Microsystems. Java RMI specification, October 1998.

[80] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1), March 1992.

[81] Simon Monk and Ian Sommerville. A model for versioning of classes in object-oriented databases. In *Proceedings of BNCOD 10*, pages 42–58, Aberdeen, 1992. Springer Verlag.

[82] H. Nielsen, P. Leach, and S. Lawrence. An HTTP extension framework. RFC 2774, Network Working Group, February 2000.

[83] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus: An architecture for extensible distributed systems. In *14th ACM Symposium on Operating System Principals*, Asheville, NC, 1993.

[84] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Intl. Conf. on Software Engineering*, Kyoto, Japan, April 1998.

[85] Vivek Pai et al. CoDeeN.

[86] KyoungSoo Park, Vivek Pai, and Larry Peterson. CoDeploy: A scalable deployment service for PlanetLab. `http://codeen.cs.princeton.edu/codeploy/`, 2004.

[87] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *In Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, October 2002. PlanetLab.

[88] P. Reichl, D. Thißen, and C. Linnhoff-Popien. How to enhance service selection in distributed systems. In *Intl. Conf. Dist. Computer Communication Networks—Theory and Applications*, pages 114–123, Tel-Aviv, November 1996.

[89] Joel Richardson and Peter Schwarz. Aspects: Extending objects to support multiple, independent roles. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, volume 20, pages 298–307, May 1991.

[90] Tobias Ritzau and Jesper Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, May 2000.

[91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[92] Jon Salz, Alex C. Snoeren, and Hari Balakrishnan. TESLA: A transparent, extensible session-layer architecture for end-to-end network services. In *Proc. of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[93] Twittie Senivongse. Enabling flexible cross-version interoperability for distributed services. In *Intl. Symposium on Dist. Objects and Applications*, Edinburgh, UK, 1999.

[94] Twittie Senivongse and Ian Utting. A model for evolution of services in distributed systems. In Spaniol Schill, Mittasch and Popien, editors, *Distributed Platforms*, pages 373–385. Chapman and Hall, January 1996.

[95] Lui Sha, Ragunathan Rajkuman, and Michael Gagliardi. Evolving dependable real-time systems. Technical Report CMS/SEI-95-TR-005, CMU, 1995.

[96] Michael E. Shaddock, Michael C. Mitchell, and Helen E. Harrison. How to upgrade 1500 workstations on Saturday, and still have time to mow the yard on Sunday. In *Proc. of the 9th USENIX Sys. Admin. Conf.*, pages 59–66, Berkeley, September 1995. Usenix Association.

[97] Andrea H. Skarra and Staney B. Zdonik. The management of changing types in an object-oriented database. In *OOPSLA*, pages 483–495, 1986.

[98] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proc. of the Usenix Technical Conference*, 2003.

[99] R. Srinivasan. RPC: Remote procedure call specification version 2. RFC 1831, Network Working Group, August 1995.

[100] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[101] Michael Stonebraker and Lawrence A. Rowe. The design of PostGres. In *SIGMOD Conference*, 1986. http://citeseer.ist.psu.edu/stonebraker86design.html.

[102] L. A. Tewksbury, L. E. Moser, and P. M. Melliar-Smith. Live upgrades of CORBA applications using object replication. In *IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 488–497, Florence, Italy, November 2001.

[103] Ajay Tirumala, Les Cottrell, Connie Logg, and I-Heng Mei. Disk throughputs. http://www-iepm.slac.stanford.edu/bw/disk_res.html.

[104] A. Trigdell and P. Mackerras. The rsync algorithm. Technical report, 1998. http://rsync.samba.org.

[105] Robert K. Weiler. Automatic upgrades: A hands-on process. *Information Week*, March 2002.

[106] Linda Wills et al. An open platform for reconfigurable control. *IEEE Control Systems Magazine*, June 2001.

[107] *Workshop on Dependable On-line Upgrading of Dist. Systems in conjunction with COMPSAC 2002*, Oxford, England, August 2002.

[108] Robert Wrembel. Object-oriented views: Virtues and limitations. In *13th International Symposium on Computer and Information Sciences (ISCIS)*, Antalya, November 1998.