Modular Software Upgrades for Distributed Systems

Sameer Ajmani[1], Barbara Liskov[2], and Liuba Shrira[3]

[1] Google, Inc.
[2] MIT Computer Science and Artificial Intelligence Laboratory
[3] Brandeis University Computer Science Department

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 must happen gradually, and there may be long periods of time when different nodes run different software versions and need to communicate using incompatible protocols. We present a methodology and infrastructure that make it possible to upgrade distributed systems automatically while limiting service disruption. We introduce new ways to reason about correctness in a multi-version system. We also describe a prototype implementation that supports automatic upgrades with modest overhead.

1 Introduction

Internet services face challenging and ever-changing requirements: huge quantities of data must be managed and made continuously available to rapidly growing client populations. Examples include online email services, search engines, persistent online games, scientific and financial data processing systems, content distribution networks, and file sharing networks.

The distributed systems that provide these services are large and long-lived and therefore will need changes (upgrades) to fix bugs, add features, and improve performance. Yet while a system is upgrading, it must continue to provide service to users. This paper presents a flexible and modular *upgrade system* that enables distributed systems to provide service during upgrades. We present a new methodology that makes it possible to upgrade distributed systems while minimizing disruption and without requiring all upgrades to be compatible.

Our system is designed to satisfy a number of requirements. To begin with, upgrades must be easy to define. In particular, we want *modularity*: to define an upgrade, the upgrader must understand only a few versions of the system software, e.g., the current and new versions.

In addition, we require *generality*: an upgrade should be able to change the software in arbitrary ways. This implies that the new version can be *incompatible* with the old one: it can stop supporting legacy behavior and can change communication protocols. Generality is important because otherwise a system must continue to support legacy behavior, which complicates software and makes it less robust. Our approach allows legacy behavior to be supported as needed, but in a way that avoids complicating the current version and that makes it easy to retire the legacy behavior when the time comes.

A third point is that upgrades must be able to retain yet *transform* persistent state. Persistent state may need to be transformed in some application-dependent way, e.g., to move to a new file format; and transformations can be costly, e.g., if the local file state is large. We do not attempt to preserve volatile state (e.g., open connections) because upgrades can be scheduled (see below) to minimize inconvenience to users of losing volatile state.

A fourth requirement is *automatic deployment*. The systems of interest are too large to upgrade manually (e.g., via remote login). Instead, upgrades must be deployed automatically: the upgrader defines an upgrade at a central location, and the upgrade system propagates and installs it on each node.

A fifth requirement is *controlled deployment*. The upgrader must be able to control when nodes upgrade. Reasons for controlled deployment include: allowing a system to provide service while an upgrade is happening, e.g., by upgrading replicas in a replicated system one-at-a-time (especially when the upgrade involves a time-consuming persistent 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 nodes being upgraded is light.

A sixth requirement is *continuous service*. Controlled deployment implies there can 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.

Our system provides an upgrade infrastructure that supports these requirements. We make two main contributions. Ours is the first approach to provide a complete solution for automatic and controlled upgrades in distributed systems. It allows upgraders to define *scheduling functions* that control upgrade deployment, *transform functions* that control transforming persistent state, and *simulation objects* that enable the system to run in mixed mode. Our techniques are either entirely new, or are major extensions of what has been done before. We support all schedules used in real systems, and our support for mixed mode improves on what is done in practice and is more powerful than earlier approaches based on wrappers [12, 24, 29], which support only a very restricted set of upgrades.

Second, our approach provides a way to understand and specify mixed mode. In particular, we address the question: what should happen when a node runs several versions at once, and different clients interact with the different versions? We address this question by defining requirements for upgrades and providing a way to specify upgrades that enables reasoning about whether the requirements are satisfied. The specification captures the meaning of executions in which different clients interact with different versions of an object and identifies when calls must fail due to irreconcilable incompatibilities. The upgrade requirements and specification technique are entirely new.

We have implemented a prototype, called Upstart, that automatically deploys upgrades on distributed systems. We present results of experiments that show that our infrastructure introduces only modest overhead, and therefore our approach is practical. We also discuss the usability of our approach in the context of several upgrades we have implemented and run.

The remainder of the paper is organized as follows. Section 2 presents an overview of our approach. Section 3 describes how to specify upgrades. Sections 4–6 discuss the three core components of our approach; Section 7 presents an example upgrade. Section 8 evaluates the overhead of our prototype, Section 9 discusses related work, and Section 10 concludes. A more detailed discussion of the approach can be found in a technical report [1].

2 Overview

This section presents an overview of our methodology and infrastructure.

We model a distributed system as a collection of objects. An object has an identity, a type that defines its behavior, and a state; it is an instance of a class that defines how it implements its type. Objects communicate by calling one another's methods (e.g., via RPC [27]); extending the model to general message-passing is future work. A portion of an object's state may be persistent. A node may fail at any point; when it node recovers, the object reinitializes itself from the persistent portion of its state.

To simplify the presentation, we assume each node runs a single top-level object that responds to remote calls. Thus, each node runs a top-level class—the class of the top-level object. Upgrades are limited to replacing top-level classes: we upgrade entire nodes at once. The top-level object may of course make use of other objects on its node to respond to requests, and an upgrade will also contain new code for these lower-level objects. We could extend this model to allow multiple top-level objects per node, in which case each could be upgraded independently.

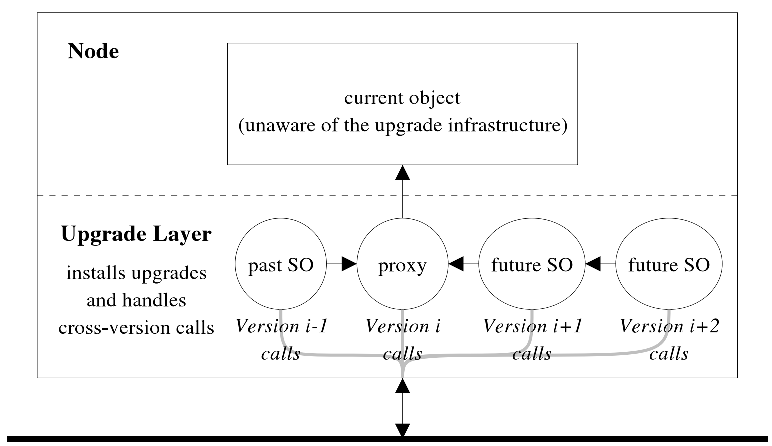

Fig. 1. *The structure of a node.*

An upgrade moves a system from one version to the next by specifying a set of *class upgrades*, one for each (top-level) class that is being replaced. The initial version has version number one (1) and each subsequent version has the succeeding version number.

A class upgrade has six components: ⟨*oldClass, newClass, TF, SF, pastSO, futureSO*⟩. *OldClass* identifies the class that is now obsolete; *newClass* 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 one. *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. These components can be omitted when not needed.

The effect of an upgrade is (ultimately) to cause every node running an object of an old class to instead run an object of the new one. We could add *filters* to the model that would determine some subset of nodes that need to upgrade. Adding filters is enough to allow restructuring a system in arbitrary ways. Of course it is also possible (without using upgrades) to add new nodes to a system and to initialize them to run either existing classes or entirely new ones.

2.1 How an Upgrade Happens

Our system consists of an upgrade server, an upgrade database, and upgrades layers at the nodes. The *upgrade server* provides a central repository of information about upgrades, and the *upgrade database (UDB)* provides a central store for information about the upgrade status of nodes. Each node runs an *upgrade layer (UL)* that installs upgrades and handles cross-version calls; the UL also maintains a local database in which it stores information about the upgrade status of nodes with which this node has communicated recently.

The structure of a node is shown in Figure 1. The node's *current version* identifies the most recently installed upgrade (or the initial version); the node's *current object* is an instance of its *current class*, which is the new class of this upgrade. The node may also be running a number of simulation objects: *future SOs* to simulate versions not yet installed at the node, and *past SOs* to simulate versions that are older than the current version.

Past and future SOs are typically implemented using *delegation*: they call methods of the object for the next or previous version, which may be the current object or another SO. These calls all move toward the current object, as shown in Figure 1.

A node's UL labels outgoing calls with the version number of the caller: 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 UL dispatches incoming calls by looking at their version number and sending them to the local object that handles that version number.

Nodes learn about upgrades because they receive a call from a node running a later version, through periodic communication with the upgrade server, or via gossip: nodes gossip with one another periodically about the newest version and their own status, e.g., their current version number and class.

When the UL learns of a newer version, it communicates with the upgrade server to download a small upgrade description. Then it checks whether the upgrade affects it, i.e., whether the upgrade contains an old class that is running at the node. (A node might be several versions behind, but it can process the upgrades one-by-one.) If the node is affected, the UL fetches the class upgrade components that concern it; drains any currently-executing RPCs; then starts a future SO if necessary, e.g., if the new type is a subtype of the old one, or if the upgrade is incompatible.

Next, the upgrade layer invokes the class upgrade's scheduling function, which runs in parallel with the node's other processing. The scheduling function notifies the UL when it is time to upgrade.

To upgrade, the UL restarts the node and runs the transform function to convert the node's persistent state to the representation required by the new class. After this, the UL does "normal" node recovery, during which it creates the current object and the SOs. Because SOs delegate toward the current object, the UL must create them in an order that allows this. First, it creates the current object, which recovers from the newly-transformed persistent state. Then it creates any past and future SOs as needed, in order of their distance from the current object.

Finally, the upgrade layer notifies the upgrade database that its node is running the new version.

When all nodes have moved to a new version, the previous version can be retired (or this could happen on command). Information about retirement arrives in messages from the upgrade server. In response, a UL discards past SOs for retired versions. This can be done lazily, since keeping past SOs around does not affect the behavior or performance of later versions.

3 Specifying Simulation

A key contribution of our approach is that we allow simulation so that nodes running different versions can nevertheless interact. But for simulation to make sense, we need to explain what it means.

Simulation enables a node to support multiple types. It implements its *current type* using its current object; it simulates old types (of classes that it upgraded from in the past) using past SOs and new types (of classes that it will upgrade to in the future) using future SOs. Some clients interact with the node via the current type, while others interact via an older or newer type. Yet all the objects implementing these types share a single identity and thus each call needs to affect and be affected by the others. It's straightforward to define these interactions when the old and new class implement the same type, or one is a subtype of the other [19], because in these cases the types already have a relationship that defines the meaning of the upgrade. Things get interesting, however, when there is an *incompatible upgrade*: when the two types are unrelated by subtyping.

This section explains what it means to simulate correctly. We capture the effects of simulation for a particular class upgrade by defining a specification for the upgrade; the specification guides the design of the simulation objects and transform function.

Correct simulation must support reasoning about client programs, not only when they call nodes that are running their own version, but also when they call nodes that are running newer or older versions, when they interact with other clients that are using the same node via a different version, and when the client itself upgrades and then continues using a node it was using before it upgraded. Furthermore upgrades of servers should be *transparent* to clients: clients should not notice when a node upgrades and changes its current type (except that more or fewer calls may fail as discussed below). 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.

We begin by defining some requirements that an upgrade must satisfy. Clearly, we require:

Type Requirement. The class for each version must implement its type.

In particular, the class implementing a future SO must implement the new type, and a class implementing the past SO must implement the old one. This requirement 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, e.g.,

$O_1.m(args)$; $O_1.m(args)$; [version 2 introduced at server];
$O_1.m(args)$; $O_2.p(args)$; [server 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 O_N is the object with which version N clients interact. Between the introduction of version 2 and the retirement of version 1, there can be an arbitrary sequence of calls to O_1 and O_2. If the server is supporting more than two types, calls to objects of all supported types can be interleaved. Although these calls can be running concurrently, we assume they occur one-at-a-time in some serial order; we discuss concurrency in Section 4.1.

To define what happens with interleaving we require:

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

An event is a call, an upgrade, or the introduction of a version.

This requirement means method calls to a current object or SO must reflect the effects of calls made to the others. If the method is an observer, its return value must reflect all earlier modifications made via other objects; if it is a mutator, its effects must reflect all earlier modifications made via other objects, and must be visible to later calls made via other objects.

When the node upgrades and its current type changes, observations made via any of the objects after the upgrade must reflect the effects of all modifications made via any object before the upgrade. For example, if a node is running several versions of a file system, modifications to a file using one of the versions must be visible to clients using the others and must continue to be visible after the node upgrades.

Together, the type and sequence requirements can be overconstraining: it may not be possible to satisfy both of them for all possible computations. When this happens, we resolve the problem by *disallowing* calls. The system causes disallowed calls to fail (i.e., to throw a failure exception). In essence, we meet the requirements above by ruling out calls that would otherwise cause problems. However, we require:

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

In other words, we can only disallow calls to past and future SOs. The rationale is that the current object provides the "real behavior" of the node, so it should not be affected by the node's support for other versions. Another point is that the code that implements the current object need not be concerned with whether there are simulation objects also running at its node, and therefore we simplify the implementation that really matters.

Disallowing takes advantage of the fact that any RPC can fail, e.g., because of network problems, so that clients won't be surprised by such a failure.

3.1 Specifying Upgrades

Now we describe how to specify an upgrade involving two types that are unrelated by subtyping, T_{new} and T_{old}. An upgrade specification has three parts, an invariant, a mapping function, and shadow methods.

The *invariant*, $I(O_{old}, O_{new})$, relates the old and new objects throughout the computation: assuming $I(O_{old}, O_{new})$ holds when a method call on one of the objects starts, $I(O_{old}, O_{new})$ also holds when the method returns. The invariant must be *total*: 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 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). However, weaker invariants can lead to fewer disallowed methods (as discussed in Section 3.2).

The *mapping function (MF)* defines an initial state for O_{new} given the state of O_{old} when T_{new} is introduced at the node. For example, the MF from the old file system to the new one would state that the new file system contains all the old files; it would also define initial values for any new file properties. The MF must be total and must *establish the invariant*: $I(O_{old}, MF(O_{old}))$ must hold.

I tells us something about what we expect from method calls. In particular, it constrains the behavior of mutators. 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, $T_{new}.\$m$. The specification of $T_{new}.\$m$ explains the effect on O_{new} of running $T_{old}.m$. Similarly, for each mutator $T_{new}.p$, we specify a related method, $T_{old}.\$p$, that explains the effect on O_{old} of running $T_{new}.p$.

A shadow method must 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:

$$pre_m(O_{old}) \wedge I(O_{old}, O_{new}) \Rightarrow pre_{\$m}(O_{new})$$
$$pre_p(O_{new}) \wedge I(O_{old}, O_{new}) \Rightarrow pre_{\$p}(O_{old})$$

Also, shadow methods must *preserve the invariant*:

$$I(O_{old}, O_{new}) \Rightarrow I(O_{old}.m(args), O_{new}.\$m(args))$$
$$I(O_{old}, O_{new}) \Rightarrow I(O_{old}.\$p(args), O_{new}.p(args))$$

Given these constraints, we can prove that the 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).

As an example, consider a upgrade that replaces a set of colored integers with a set of flavored integers. This example is analogous to an upgrade that changes a property of files in a file system.

We begin by choosing an invariant I that we want to hold for each ColorSet (O_{old}) and FlavorSet (O_{new}). We could require that the two sets contain the same integers:

$$\{\, x \mid \langle x, c\rangle \in O_{old} \,\} = \{\, x \mid \langle x, f\rangle \in O_{new} \,\} \tag{1}$$

A stronger invariant maps colors to flavors:

$$\begin{gathered}\langle x, \text{blue}\rangle \in O_{old} \Leftrightarrow \langle x, \text{grape}\rangle \in O_{new},\\ \langle x, \text{red}\rangle \in O_{old} \Leftrightarrow \langle x, \text{cherry}\rangle \in O_{new},\\ \ldots\end{gathered} \tag{2}$$

Whereas (1) treats colors and flavors as independent properties, (2) says these properties are related. A weaker invariant allows O_{new} to contain more elements than O_{old}:

$$\{\, x \mid \langle x, c\rangle \in O_{old} \,\} \subseteq \{\, x \mid \langle x, f\rangle \in O_{new} \,\} \tag{3}$$

The next step is to define a mapping function. For invariant (1), we might have:

$$O_{new} = MF(O_{old}) = \{\, \langle x, \text{grape}\rangle \mid x \in O_{old} \,\} \tag{4}$$

As required, this MF establishes I.

Here are possible definitions of the shadow methods, assuming that both types have an insert method that adds an element with a specified color or flavor, and a delete method.

void ColorSet.$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.$delete(x)
 effects: $this_{post} = this_{pre} - \{\langle x, c\rangle\}$

void FlavorSet.$insertColor(x, c)

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

void FlavorSet.$delete(x)

effects: $\mathit{this}_{post} \;=\; \mathit{this}_{pre} - \{\langle x, f\rangle\}$

These definitions satisfy invariant (1). They do not work for invariant (2) since in that case the shadows must preserve the color-flavor mapping. Our original mapping function and shadow methods would work for invariant (3), but we could use weaker definitions, e.g., define FlavorSet.$delete to have no effect.

3.2 Disallowed Calls

There was no need to disallow any methods in the example above. But sometimes disallowing is needed.

When we specify an upgrade we implicitly define a "compound type," $T_{old\&new}$. This type has the methods of both T_{old} and T_{new}. Its objects contain the old state and the new state and they satisfy the invariant I.

The specification of a mutator is a combination of its original specification and its shadow specification provided in the upgrade; the former defines its effect on its own type, and the latter defines its effect on the other type in the upgrade. E.g., the specification of insertFlavor states its effect on the FlavorSet (its original specification) and on the ColorSet (as defined by the specification of ColorSet.$insertFlavor).

If $T_{old\&new}$ is a subtype of both the old and new types, the simulation is working properly, since users will always see the behavior they expect. In the case of the upgrade from FlavorSet to ColorSet, this subtype property holds. But sometimes it doesn't, and in this case we solve the problem by disallowing. We might disallow all calls to a method, or only some calls, based on the parameters of the call or the current state of the object.

For example, consider an upgrade that replaces GrowSet with IntSet; a GrowSet is like an IntSet except that it never shrinks because it has no delete method. The shadow of delete on a GrowSet object must remove the deleted object, assuming the invariant that the two objects have the same elements. Since GrowSet objects never shrink, we must disallow the delete method in the future SO for IntSet. However, once the node upgrades, we can no longer disallow this method since the current object is now an IntSet. Therefore the state of the past SO for GrowSet can shrink. Since this does not match the specification of GrowSet, we must disallow any GrowSet methods that would expose the problem. Thus we would need to disallow GrowSet.isIn.

Thus disallowing is done differently for the future SO and the past SO: for the future SO we only disallow methods of the new type, while in the past SO, we only disallow old type methods. These restrictions on disallowing follow from our disallow constraint: they ensure that all methods of the current object are allowed.

To disallow for the future SO, we proceed as follows. First we disallow all mutators of the new type whose shadow definitions for the old type would cause violations of the specification of the old type; this disallowing will ensure that $T_{old\&new}$ is a subtype of the old type. In addition, if the new shadows of any old type methods violate the specification of the new type, we disallow new methods that expose these violations; this ensures that users of the future SO won't notice that something strange is going on.

The situation for the old type is similar. We disallow any old methods whose shadows would cause violations of the specification of the new type; this way we will obtain a subtype of the new type. Also, if any shadows of the new type methods violate the specification of the old type, we disallow old methods that expose these violations to ensure that users won't see the odd behavior.

This notion of "exposing violations" has a different meaning for past and future SOs, because a future SO will eventually become the current object and at that point all its methods will be allowed. These calls represent another way of noticing a violation, and must be taken into account when disallowing. For example, consider the reverse upgrade (from IntSet to GrowSet). The future SO in this case must disallow both isIn and insert. It must disallow insert because once the GrowSet becomes the current object, calls of isIn will be allowed, and at that point the absence of an object that had previously been inserted into the GrowSet object would be noticed!

Weakening the invariant can reduce the need to disallow. For example, if we allowed the GrowSet object to contain a superset of the elements of the IntSet object, we would not need to disallow any methods in either the past or future SO.

In general, the upgrader should choose the weakest invariant that makes sense for the two types in the upgrade, in order to disallow as little as possible. Disallowing is unlikely to be what users want; therefore the upgrader may choose to avoid it by using an accelerated schedule for the upgrade (see Section 6).

3.3 Multiple Upgrades

The previous sections have discussed what is needed to specify and upgrade in isolation, assuming that no other upgrade is "active." In other words we considered a system that was everywhere running a particular version, and defined an upgrade to move it to the next version. Now we consider a more general case, in which more than one upgrade may be in progress.

If some upgrades are in progress when a new one is defined, and if some of those earlier upgrades are incompatible, we are in a situation where the previous upgrade is actually defining not T_{new} but in fact $T_{old\&new}$. Therefore, we need to extend our specification approach so that we define the intended behavior of these extra methods—the ones in $T_{old\&new}$ of the previous upgrade that aren't also in T_{new} of the previous upgrade. The extra methods are precisely the shadows of the mutators of the old type. (We do not need to consider the shadow definitions for the mutators of the new type because those details are handled by the previous implementation.)

Thus we need to provide shadows for these shadows. In addition, we need to use $T_{old\&new}$ from the previous upgrade when deciding what methods to disallow for the past and future SOs of the current upgrade.

As an example, suppose we define a second upgrade to follow the upgrade from ColorSet to FlavorSet. This second upgrade defines a CommentSet in which each element of the set has both a flavor and an associated comment. This upgrade is compatible since CommentSet is a subtype of FlavorSet.

However to define the upgrade we need to provide an explanation of the effect of a call on ColorSet.insertColor on the CommentSet. This is done by considering FlavorSet.$insertColor; the specification of this shadow explains the effect of running the insertColor on the FlavorSet. We provide a shadow for this method, CommentSet.$$insertColor, which explains the additional effect on the CommentSet. In this example, it isn't necessary to disallow any new methods because we have the subtype property.

One point about writing these specifications is that a kind of "transitive" disallowing is possible. Suppose the specification for the old upgrade disallows a method of the new type. Then when we shadow this method, there are two cases: either it is disallowed (because its upgrade hasn't yet been installed) or not. However, when the old method is disallowed, this necessarily implies that the new one is too. Therefore we require that the shadow specification only explain what happens when the method being shadowed is allowed.

The key question about specifying upgrades when many upgrades are in progress is modularity: how much does an upgrader need to know to specify an upgrade? Clearly the upgrader must know the old and new types of the current upgrade *plus* the specification of the earlier upgrade. However, this earlier upgrade has both an old and new type, and it's possible that in order to understand its specification it is necessary to understand both of them. Fortunately, this appears to not be necessary most of the time because the shadows of T_{old} methods are usually specified in terms of the T_{new} state; in this case the definer of the next upgrade need not understand T_{old}. The CommentSet example is like this, and so are all the real examples we looked at; the only ones that aren't are pathological examples we invented.[4]

If a pathological example were to arise, it may be possible to avoid the problem by changing the invariant. Otherwise it may be necessary to go arbitrarily far back in the chain of "active" upgrades (ones whose old type has not yet been retired). To avoid this, the upgrader might decide to use an eager schedule for the upgrade to limit the time during which defining future upgrades requires understanding of the old type.

4 Implementing Simulation

This section presents ways to use simulation objects to implement multiple types. The approaches differ in how calls are dispatched to objects (i.e., which objects implement which types) and how simulation objects can interact with one another. The first "direct"

[4] An example that causes problems is the following. The old upgrade replaces ColorSet with FlavorSet, but the invariant specifies some function f that maps colors to flavors, where several colors map to the same flavor. Furthermore the specification of ColorSet.setColor states that the color of an item in the set can be changed only when its current color is blue. To define the shadow FlavorSet.$setColor, we need to consult the state of the ColorSet object to determine the current color of the item, since only then will we know what its flavor will be:

void FlavorSet.$setColor(x, c)

effects: $\langle x, blue\rangle \in prev.this_{pre} \Rightarrow this_{post} = this_{pre} - \{\langle x, *\rangle\} \cup \{\langle x, \mathrm{f(c)}\rangle\}$

approach is simple and is similar to what others have proposed [12, 24, 29]. However it lacks expressive power, and therefore we instead use a much more powerful "interceptor" approach.

4.1 Direct Approach

In the *direct approach*, calls for each version are dispatched *directly* to the object that implements the type for that version. Each SO implements just its own type and can delegate calls to the next object closer to the current object: the next older object for future SOs, the next newer object for past SOs. When an upgrade is installed, a past SO for the old type is created if necessary (i.e., if the new type isn't a subtype of the old type). Figure 2 depicts how SOs are managed in the direct approach.

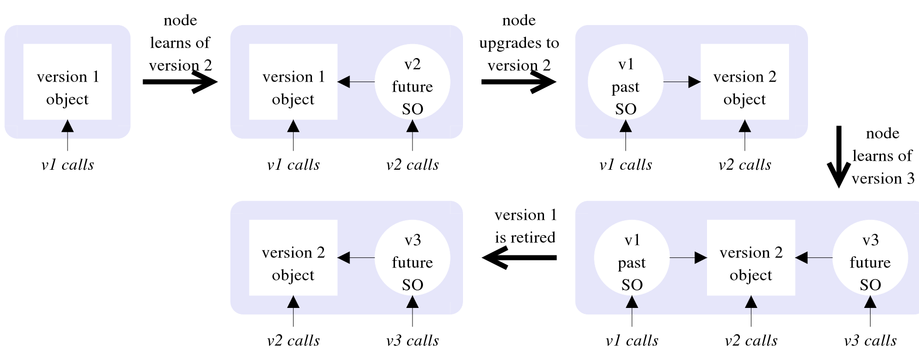

Fig. 2. *The direct approach, presented as a sequence of states of a node. Large arrows are state transitions. In each state, the box is the current object, and the circles are SOs. Objects may delegate calls as indicated by the small arrows. Each object handles calls only for its own version.*

The direct approach is simple but has limited expressive power. The most serious problem is that there is no way for an SO to be informed about calls that go directly to its delegate, and as a result it can do the wrong thing. 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 associated colors, which it updates when it runs its own methods. However, consider the following sequence of calls (here O refers to the SO's delegate): 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 default color, e.g., "blue", as specified for the shadow of IntSet.insert(x).

Since we cannot prevent the SO state from being stale, our only recourse is to disallow SO methods (we cannot disallow O.delete because of the disallow constraint). It may seem that we must disallow SO.getColor, since it is the method that revealed the problem in our example, but in fact we must disallow SO.insertColor because otherwise we'll be able to observe the problem when the upgrade is installed (since at that point calls to the getColor will be allowed). And disallowing SO.insertColor is sufficient; we needn't disallow SO.getColor in addition (because every integer is blue).

A second problem is that the direct approach provides no way for the different versions to synchronize. Since calls go directly to the different versions, SOs have no way to control how calls are applied to their delegates. 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. With the direct model, how can the future SO ensure that two adjacent items are dequeued, since a client could call deq directly on the delegate while the SO is carrying out deq2?

It does not work for the upgrade layer to force methods to execute one-at-a-time, as this may cause the distributed system to deadlock. Instead, the delegate might provide some form of application-level concurrency control, such as 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 delegator can use lockdeq to implement deq2 correctly. This solution is complex, however. Furthermore, if the delegate does not provide appropriate concurrency control methods, the upgrader's only choice is to disallow deq2.

4.2 Interceptor Approach

The interceptor approach avoids the problems of the direct approach.

In the *interceptor approach*, the simulation object for the latest version handles all calls (it *intercepts* calls intended for the earlier versions). The upgrade layer dispatches all calls for any version to the newest SO, which executes the calls by delegating to the preceding object, which may be the current object or another SO.

If the current upgrade is compatible, then when the upgrade occurs, the node replaces its current object and the future SO with an instance of the new class, which becomes the current object of the node. The current object continues to handle all calls intended for its predecessor. There is no need for a past SO, because calls made by clients running at the old version are handled by the current object.

However, when the current upgrade is incompatible, the current object isn't sufficient since we want it to implement only the new behavior, and therefore it isn't prepared to handle calls for the old type of its upgrade. Therefore in this case, the upgrade replaces the future SO and current object with an instance of the new class *and* past SO. Furthermore all incoming calls are dispatched to the past SO, which simulates the old type's behavior and delegates to the current object. Figure 3 illustrates this approach.

If another upgrade is introduced, it receives a future SO, which must be prepared to handle the methods of the new type, the old type, *and* the old type of the previous upgrade. The future SO handles these methods by delegating to the past SO of the previous upgrade; because of this delegation, handling these extra calls isn't a burden.

This situation continues until the old type of the incompatible upgrade is retired. At this point the past SO can be removed and calls that used to be delegated to it will go directly to the current object. The calls won't be to methods of the old incompatible type, since that type is no longer in use.

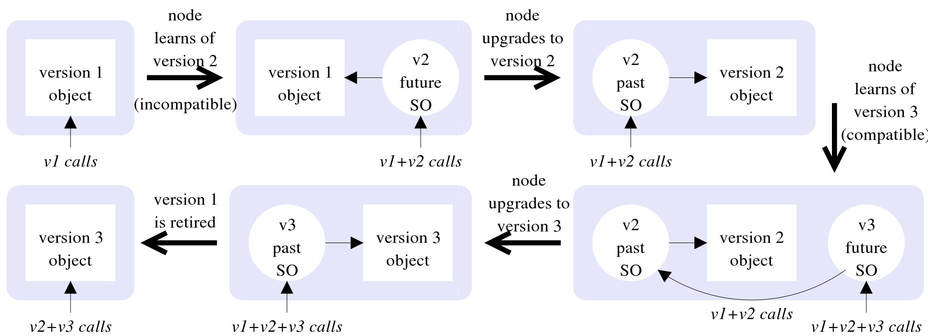

Fig. 3. *The interceptor approach, presented as a sequence of states of a node. Large arrows are state transitions. In each state, the box is the current object, and the circles are SOs. Objects may delegate calls as indicated by the small arrows. One object in each state intercepts all calls.*

The interceptor model works well as long as there is only one active incompatible upgrade. However, this model has only one past SO object in existence at any time, and this object must handle all the legacy behavior. It can do this by using the previous past SO as a subobject, which can delegate to the current object *if* the upgrade that just happened is compatible. Otherwise the new past SO will have to do more of the work of simulating past behavior.

Therefore a good upgrade strategy is to always retire an incompatible upgrade before introducing the next incompatible upgrade. We believe this is a reasonable approach since incompatible upgrades are introduced relatively infrequently.

4.3 Implementing SOs in the Interceptor Approach

Now we consider what is needed to implement SOs.

Obviously the implementation needs to satisfy the specification: the future SO needs to implement $T_{old\&new}$ of its upgrade with disallowing of T_{new} methods, while the past SO needs to implement $T_{old\&new}$ with disallowing of T_{old} methods.

These implementations must cause modifications of the state of the SO itself, but they must also do the right thing on other versions. For example, when the future SO handles a call on a mutator, it must also mutate its predecessor in the version chain. In our approach this is always done via delegation: an SO will call one or more methods on its predecessor (or successor if it is a past SO).

One interesting point is that the delegation may fail because that call is disallowed. When this happens, the call to the delegating object must also be disallowed. However, what happens due to disallowing can change during the lifetime of a future SO. Initially a call to the predecessor might be disallowed because it is a call to a method of the new type of the earlier upgrade, and upgrade hasn't happened yet. However, once that upgrade has happened, the call will be allowed, and therefore the call on the future SO should also be allowed. Thus the implementations in the future SO will typically be written to disallow if any disallowed calls are encountered, and to allow otherwise.

Each call that arrives from a client must be executed *atomically* at each object in the chain, and if some delegated call fails (whether because it is disallowed, or for some other reason), the states of all objects must be left unchanged (i.e., the call aborts). This can sometimes be tricky to ensure. For example, suppose that to carry out a call on method *m* of the future SO, two calls, to $p1$ and $p2$, are made to the predecessor object, where $p1$ is a mutator. The implementation in the future SO must be done in such a way that if the call to $p2$ is disallowed, the state of the predecessor doesn't change. This can be accomplished by checking in advance whether the call to $p2$ will succeed, assuming such a method exists. The method will exist if the old type of the upgrade is "complete" [16]; if not, it may sometimes be necessary to add extra observers to the predecessor to permit more access to its state. For example, if $p1$ is *insert(x)* and $p2$ is *remove(y)*, it may be necessary to check *isIn(y)* before calling $p1$ then $p2$.

A future SO comes into existence when the infrastructure at its node first learns about the upgrade. The node drains its currently-executing RPCs [26], and then creates the SO by running a default constructor. This code has no access to any arguments, nor can it access the object implementing the old version. Therefore it is unlikely to be able to fully implement the mapping function; instead it must leave the object in a partially-initialized state, and methods that are called after this point complete the initialization (e.g., by making calls on the delegate). This limitation on how an SO initializes is intentional so that SO installation can be a lightweight (and fast) operation.

5 Transform Functions

A transform function (TF) reorganizes a node's persistent state from the representation required by the old instance and future SO to that required by the new instance and past SO. It must implement the *identity mapping*: the post-TF abstract state of the past SO is the same as the pre-TF state of the old object, and the post-TF abstract state of the new object is the same as the pre-TF state of the future SO. Thus, clients 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 TF must be *restartable*, because the node might fail while the TF is running. If this happens, the upgrade infrastructure simply re-runs the TF, which must recover appropriately.

A TF may not call methods on other nodes, because we can make no guarantees about when one node upgrades relative to another, so other nodes may not be able to handle the calls a TF might make. This restriction does not limit expressive power; if a node needs to recover state from another node (e.g., in a replicated system), it can transfer this state *after* it has completed the upgrade. This restriction helps avoid deadlocks that may occur if nodes upgrading simultaneously attempt to obtain state from each other. It also makes TFs simpler to implement and reason about.

6 Scheduling Functions

Scheduling functions (SFs) allow an upgrader to control upgrade progress. SFs run on the nodes themselves, so they can consider the node's state in deciding when to upgrade. But often what's more important for SFs is the state of the system; in particular, the upgrade state of other nodes. Therefore we provide SFs with additional information: a central upgrade database (UDB) that records the upgrade status of every node and can contain user-defined tables (e.g., that authorize the upgrades of subsets of nodes), and per-node local databases (LDBs) that record information about the status of other nodes with which a node communicates regularly. Each class upgrade has its own scheduling function, which allows the upgrader to consider additional factors, such as the urgency of the class upgrade and how well the SOs for that class upgrade work.

When defining an SF, the first priority is to ensure that all nodes eventually upgrade. We guarantee this trivially by requiring that the upgrader specify a timeout for each SF.

The second priority is to minimize service disruption during the upgrade. How this is accomplished depends on how the system is designed. For example, Brewer [7] describes several upgrade schedules used in industry; each of these can be implemented easily as scheduling functions:

- A *rolling upgrade* causes a few nodes to upgrade at a time; this makes sense for replicated systems and can be implemented by an SF that queries its local database to decide when its node should upgrade, e.g., by waiting its turn in a sequence.
- A *big flip* causes half the nodes in a system to upgrade at once; this makes sense for systems that need to upgrade quickly and can be implemented by an SF that flips a coin to decide whether its node should be in the first or second upgrade group.
- A *fast reboot* causes all nodes to upgrade at once; this make sense when cross-version simulation is poor and can be implemented by an SF that causes its node to upgrade at a particular wall-clock time. Alternatively, this SF could wait for an explicit signal written to the UDB or sent via RPC.

The implementations of these SFs are each just a few lines of script.

A variety of other schedules are possible, e.g., "wait until the node's servers upgrade," "wait until all nodes of class C upgrade," "wait until the node is lightly loaded," and "avoid creating blind spots in the sensor network." Some of these schedules require centralized knowledge, which is provided via the UDB; others require local knowledge, which is provided via the node's state and LDB. Our goal is to provide sufficient flexibility so that upgraders can build a library of SFs according to the needs of their system; once this is done, an upgrader simply selects an SF for each class upgrade from the library.

Upgrade schedules can help the upgrader avoid implementing difficult SO features. For example, it may be impractical to simulate a certain method of a new server type. We can avoid the need to simulate this method by scheduling the upgrade such that servers upgrade to the new type before any clients upgrade; thus, the difficult-to-simulate method will not be called until the servers have upgraded.

An upgrader may want to test an upgrade on a few nodes and, if those upgrades fail, roll them back and abort the remaining upgrades. This policy is implementable with

SFs (by recording upgrade failure in the UDB), though we do not discuss the details of how to rollback the failed upgrades here.

7 Example

In developing our methodology we looked at many examples, focusing on incompatible upgrades and real distributed systems including Thor [18], NFS [8], and DHash [9]. Some of the upgrades were ones that had actually happened, while others were invented. Our goal was to come up with challenging examples so that we could make sure our approach had sufficient expressive power, and so that we could understand the challenges in specifying upgrades and implementing SOs.

In this section we present a brief example of an incompatible upgrade to illustrate our approach. The example is a challenging one because the old and new types are quite different and there are several ways to resolve the differences. The upgrade replaces a file system that uses Unix-style permissions with one that uses per-file access control lists (ACLs) [15]. We assume the file system is distributed: the files are stored at many servers. The upgrade contains two class upgrades: one for clients (to switch to using ACLs) and one for servers (to switch to providing ACLs).

We assume there is no particular order in which nodes upgrade; thus clients might be ahead of servers and vice versa. A possible schedule might have a client SF that waits until the client is idle, while the server SF upgrades servers round-robin over some extended time period.

Each file in the old system has read, write, and execute bits for its owner, its group, and everyone else (the "world"). Thus, the old state (O_{old}) is a set of tuples:

$$\langle \mathit{filename, content, owner, or, ow, ox, group, gr, gw, gx, wr, ww, wx} \rangle$$

Only the owner of a file can modify the file's permissions, group, or owner. The new state (O_{new}) is a set of

$$\langle \mathit{filename, content, acl} \rangle$$

tuples, where *acl* is a sequence of zero or more $\langle \mathit{principal, r, w, x, a} \rangle$ tuples. Principals with the *a* permission are allowed to modify the ACL.

There are many invariants one could imagine for this example. Our invariant $I(O_{old}, O_{new})$ is very weak:

$$\begin{aligned}
&\langle \mathit{filename, content, owner, or, ow, ox, group, gr, gw, gx, wr, ww, wx} \rangle \in O_{old} \\
&\Leftrightarrow (\langle \mathit{filename, content, acl} \rangle \in O_{new} \\
&\quad \wedge (\langle \mathit{owner, or, ow, ox}, \text{“}\mathit{true}\text{”} \rangle \in \mathit{acl} \vee (\mathit{owner} = \text{“}\mathit{nobody}\text{”} \wedge \neg\, \mathit{or} \wedge \neg\, \mathit{ow} \wedge \neg\, \mathit{ox})) \\
&\quad \wedge (\langle \mathit{group, gr, gw, gx}, \text{“}\mathit{false}\text{”} \rangle \in \mathit{acl} \vee (\mathit{group} = \text{“}\mathit{nobody}\text{”} \wedge \neg\, \mathit{gr} \wedge \neg\, \mathit{gw} \wedge \neg\, \mathit{gx})) \\
&\quad \wedge (\langle \text{“}\mathit{system{:}world}\text{”}, \mathit{wr, ww, wx}, \text{“}\mathit{false}\text{”} \rangle \in \mathit{acl} \vee (\neg\, \mathit{wr} \wedge \neg\, \mathit{ww} \wedge \neg\, \mathit{wx}))
\end{aligned}$$

This invariant says that each file in O_{old} is in O_{new} with the same contents, and either the owner of the file in O_{old} appears in the ACL in O_{new} with the same permissions plus the ACL-modify permission, or the owner is the special user "nobody" and the owner permissions are all false, and similarly for the group and world permissions (except these have no ACL-modify permission). We need to include the "nobody" case so that I is total, i.e., so there is a defined state of O_{old} for each state of O_{new}, and vice versa (in particular, consider the case when the ACL is empty). Clearly other invariants are possible, e.g., to select a particular owner among several in the ACL to be the owner in the permissions.

The mapping function for this upgrade states that each file in O_{new} has the same contents as in O_{old} and an ACL containing the owner, group, and world permissions from O_{old}. The initial ACL grants ACL-modify permissions only to the owner.

The shadow methods must preserve I. When a client modifies a file in O_{old}, that file is also modified in O_{new}, and vice versa. Furthermore, the file system must only allow file operations that are consistent with the file's permissions (in the old system) or ACL (in the new system). But consistency is a problem, since ACLs are more expressive than permissions.

Let's consider the case of the future SO first. If the future SO allows modifications of ACLs, clients of the permissions system may see modifications made by clients of the new system that do not appear to have the correct permissions. For example, if an owner in the ACL system adds as a second owner a user of the permissions system, and later removes that user as an owner, a client using the permissions system and running as that user might notice odd behavior.

To prevent this, we might disallow such operations in the future SO. However, we cannot disallow modifications of ACLs once the server has upgraded, which means that we must figure out what to do for users of the permissions systems when such changes happen. A possible solution is to make it impossible for users of the permissions system to notice odd behavior by not allowing them to do anything at all. But this doesn't seem like a good idea: clearly we don't want to prevent users of the permissions system access to files. A second possibility is to disallow only cases where observation of odd behavior is possible. For example, we might disallow access only for files where there is more than one owner. This second solution is less draconian than the first but still seems undesirable.

In general when defining an upgrade it may not be possible to allow all behavior, and furthermore, almost always disallowing isn't desirable. In this particular example, however, we have an out because file systems don't guarantee that owners are in complete control, since the superuser can change anything: the specification of a file system does not rule out the kinds of odd behavior discussed above. Therefore we can in fact allow all methods in both the past and future SO.

Now let's consider how to implement the past and future SOs. Implementing the past SO is easy: it just needs to present the permissions corresponding to the ACLs in O_{new} and map any permissions modifications to the appropriate ACL modifications.

The implementation of the future SO is trickier. If it allows ACL mutations without restrictions it must keep track of all the entries in each ACL, not just the ones that map to permissions in O_{old} (O_{new} may be more permissive than O_{old} because of these extra ACL entries). Furthermore, it would need to run with superuser privileges in order to support the behavior in the ACL, which may be undesirable. Therefore the upgrader might choose to disallow the creation of ACLs via the future SO that have entries with no corresponding permissions in O_{old}.

The effort to implement the SOs is modest. SOs need to provide the extra behavior needed at that version, e.g., to store the extra information in the ACL for the future SO; the rest of the work is delegated. Furthermore, what is happening in the SO is similar to what will happen in the version it is simulating, once that becomes the current version, and therefore this code can be used in implementing the SO. For example, all the code for manipulating ACLs is available when the future SO for this upgrade is implemented.

The TF must produce the state of O_{new} (files and ACLs) from that of O_{old} (files and permissions) and the future SO (if it has state). Therefore, if we decide to allow unrestricted ACLs creation in the future SO, the TF would need to access to its state to create the current object.

The exact choice of what to allow is up to the definer of the upgrade, and as this example shows, there may be several possible choices. Furthermore, the decision might take into account implementation difficulties: the upgrader might choose to disallow some behavior because it would be difficult to implement.

8 Evaluation

This section evaluates Upstart, our prototype upgrade infrastructure. The purpose of this prototype is to demonstrate that our methodology can be realized efficiently, not to advocate any particular implementation. We describe Upstart, the results of microbenchmarks, and our experience running a distributed upgrade.

Upstart implements the upgrade server as an Apache web server. The upgrade server stores upgrade descriptions and code for upgrades. The upgrade descriptions are small; they identify the new code using URLs. To reduce load on the upgrade server, we use the Coral content distribution network [11] to cache and serve the code.

Upstart implements the upgrade database (UDB) as a PostGres database that resides on the upgrade server. Nodes append new records to the UDB periodically but do not write to the UDB directly, as this would cause too much contention in a large system. Instead, nodes send their header over UDP to a `udb_logger` process that in turn inserts records in the UDB. Under heavy load, some headers may be lost; but this is okay, as nodes will periodically resend updated headers.

The upgrade layer runs on each node, in a separate process from the application. This separation is important: if the application has a bug (e.g, that causes it to loop forever), the upgrade layer must be able to make progress so that it can download and install code that fixes the bug. The UL fetches upgrades from the upgrade server, runs the SF (in a separate process), runs SOs, installs upgrades, and writes status information to the UDB. Once a minute, the UL piggybacks headers on the messages it sends to other nodes it has communicated with lately to inform them of its status. Each UL maintains status information in a local PostGres database (LDB); scheduling functions can query the LDB to make scheduling decisions. To avoid writing to the LDB on the critical path, the UL passes headers to a local `udb_logger` process.

The UL is implemented as a TESLA handler [23]. TESLA is a dynamic interposition library that intercepts `socket`, `read`, and `write` calls made by an application and redirects them to *handler* objects. When the application creates a new socket, TESLA creates an instance of the UL handler. When the application writes data to the socket or when data arrives on that socket from the network, TESLA notifies the UL via method calls. Since TESLA is transparent to the application, the application can listen on its usual port and communicate normally, which is important for applications that exchange their network address with other nodes, such as peer-to-peer systems.

We implemented the UL and SOs in event-driven C++. To reduce the implementation burden on the upgrader, we provide code-generation tools that simplify the process of implementing SOs for systems that use Sun RPC [27]. Providing support for other kinds of systems is straightforward and requires no changes to the upgrade infrastructure.

8.1 Microbenchmarks

The most important performance issue is the overhead imposed by the upgrade layer when no upgrades are happening, as this is the common case. This section presents experiments that measure these overheads and show them to be modest.

We ran the experiments with the client and server on the same machine (connected over the loopback interface) and on separate machines (connected by a crossover cable). Each machine is a Dell PowerEdge 650 with four 3.06 GHz Intel CPUs, 512 KB cache, 2 GB RAM, and an Intel PRO/1000 gigabit ethernet card. We also ran experiments on the Internet; we do not report the results here, as the latency and bandwidth constraints of the network dwarf the overhead of the upgrade infrastructure.

In each experiment we ran a benchmark and compared its baseline performance with the costs imposed by our system. In the graphs, *Baseline* measures the performance of the benchmark alone. *TESLA* measures the performance of the benchmark running with the TESLA "dummy" handler on all nodes; it adds the overhead for interposing between the benchmark and the socket layer, context switching between the benchmark and the TESLA process, and copying data between the benchmark to the TESLA process. *Upstart* measures the performance of the benchmark running with the upgrade layer on all nodes; it adds the overhead for adding/removing version numbers on messages and bookkeeping in the proxy object. In our experiments, we disabled upgrade server polling and periodic header exchanges. In our prototype, prepending a version number to a message requires copying the message to a new buffer; so each RPC incurs two extra copies. These copies could be avoided by extending TESLA to support scatter-gather I/O.

Table 1 summarizes the results.

	Null RPC (loopback)			Null RPC (crossover)			100MB TCP transfer		
	5%	50%	95%	5%	50%	95%	5%	50%	95%
Baseline	50μs	51μs	53μs	247μs	382μs	769μs	896ms	896ms	923ms
TESLA	128μs	139μs	154μs	371μs	382μs	782μs	896ms	898ms	919ms
Upstart	192μs	206μs	223μs	245μs	388μs	819μs	897ms	908ms	936ms

Table 1. *Microbenchmark results (N=100000 for Null RPC, N=100 for TCP). For each experiment, the 5th, 50th (median), and 95th percentile latencies are given.*

In the Null RPC benchmark, a client issues empty RPCs to a server one-at-a-time using UDP. By instrumenting the code with timers, we found that the time spent in the client and server ULs is approximately equal, which is as expected since each side sends and receives one message per RPC. Half the time in the UL is spent in the proxy objects, and the other half is spent adding and removing version numbers.

Over the loopback interface, the latencies are normally distributed; but over the crossover cable, we see significant variance. This is due to *interrupt coalescing* done by the gigabit ethernet card, in which the card and/or driver delay interrupts so that one interrupt can be used for multiple packets. A cumulative distribution function of this data (not shown) reveals that the latencies cluster at 125μs intervals; this accounts for the fact that TESLA's 5th percentile is close to the median value.

In the TCP benchmark, a client transfers 100 MB of data to a server using TCP (without RPCs) over a crossover cable. The upgrade layer sees the 100 MB transfer as 12,800 8 KB messages (8 KB is the block size in the benchmark). The UL overhead is due to copying these messages and adding/removing version numbers.

8.2 Experience

To evaluate Upstart "in the field," we defined and ran a simple upgrade on PlanetLab, a large research testbed [21]. Specifically, we deployed DHash [9], a peer-to-peer storage system, on 205 nodes and installed a null upgrade on it. We chose a null upgrade to isolate the effect of the upgrade infrastructure on system performance and behavior.

Defining the upgrade was straightforward: no TF or SOs were required. The SF 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 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. The upgrade ran as expected, and the DHash network remained functional throughout.

We also ran an experiment to evaluate the effect of an upgrade on DHash client performance. Here the system consists of four nodes, each running a DHash server; one node also ran the DHash client. Before the upgrade began, we stored 256 8KB data blocks in the system. The client fetches the blocks one-at-a-time in a continuous loop and logs the latency of each fetch. Figure 4 depicts the fetch latencies over the course of the experiment.

The three non-client nodes upgrade round-robin, two minutes apart. The TF causes an upgrading node to sleep for one minute. Figure 4 reveals a stutter in client performance when each node goes down, but the client fetches resume well before each node recovers. The fetch performance while one node is down is slightly less than when all nodes are up.

The precise effect of an upgrade on clients depends somewhat on the application. With better timeouts, for example, the DHash client may see less stutter when nodes fail. Furthermore, we expect the client to see very little stutter in a larger system, as clients are less likely to need to access a node that is upgrading.

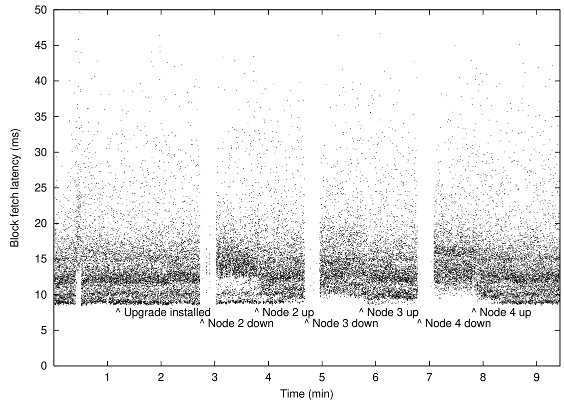

Fig. 4. *DHash block fetch performance during an upgrade.*

9 Related Work

Distributed upgrades have been explored in systems with a wide variety of requirements, some similar, some different from ours. We compare our approach to the related work in research systems and to the current practice in real-world Internet systems.

9.1 Research on Upgrades

Reconfigurable distributed systems [2, 4, 5, 14, 17, 22] support the replacement of subsystems for specific distributed object systems, provided the new type implemented by a subsystem is compatible with the old one. These approaches do not support incompatible upgrades, and they stall when nodes in the subsystem fail.

A few systems support cross-version interaction using wrappers: PODUS [12] supports upgrades to individual procedures in a (possibly distributed) program, and the Eternal system [29] supports upgrades for replicated CORBA objects. But these systems do not consider the correctness issues of cross-version interoperation. Moreover, they use a weaker implementation model than Upstart since they do not allow chaining of wrappers and therefore do not meet our modularity requirement.

The closest approach to ours is Senivongse's "evolution transparency" approach [24], which uses chained *mapping operators* to support cross-version interoperation in a modular way. However, this work does not provide a correctness model: it does not define what system behavior clients can expect after they upgrade or when they communicate with clients running different versions.

Many of the correctness issues that arise in upgrading distributed systems also arise in schema evolution for object-oriented databases, where one object calls 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 [6]; others [20,25] use mixed mode: they allow objects of different versions to interact but do not consider general consistency issues. Encore [25] supports cross-version calls for a limited class of version changes via a version set interface that is a union of all the versions of that type. The work on views in O2 [3] provides a comprehensive study of how mutations made to one object 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. However, whereas a database can use schema information to detect correctness violations and reject mutations dynamically, the SO implementor must determine which calls to disallow statically.

Finally, we consider the state preservation requirement. The goal of *dynamic software updating* [10, 13, 26, 28] is to enable a node to upgrade its code and transform its volatile state without shutting down. These techniques require implementor to identify where in the program reconfiguration can take place and are typically language-specific. Furthermore, these points must guarantee that no future execution threads will reference the old types; this can be achieved either by draining the old threads [26] or by detecting such points statically [28]. Our approach guarantees this property by shutting down a node before changing its code. Dynamic updating is complementary to our approach and could be used to reduce downtime during upgrades.

9.2 Real-World Upgrades

Internet and web service providers must upgrade large-scale distributed systems regularly. How they do so depends on whether the upgrade is internal to the service or externally visible to clients and whether the upgrade is compatible or incompatible.

For web services, upgrades are either internal to the service or, if they are externally-visible, are usually compatible. Furthermore, it's acceptable for some clients of a service to see new behaviors while others see the old ones, which means its client-facing nodes may upgrade gradually.

Internally, Internet services are tiered: the topmost tier faces clients; middle tiers implement application logic; and the bottommost tiers manage persistent state. Internal upgrades change the code of one or more tiers and may change the protocols between them. Compatible upgrades are straightforward: the lowermost affected tier is restarted using a rolling upgrade [7], then the next-lowermost tier is upgraded, and so on up the stack. Since the upgrade is compatible, calls made by higher tiers can always be handled by the lower tiers.

Incompatible, internal upgrades are typically executed by upgrading datacenters round-robin: drain a datacenter of all traffic (and redirect its clients to other datacenters), upgrade all its nodes, warm up the datacenter, restore its traffic, then repeat for the next datacenter. Thus nodes in the same datacenter never encounter incompatibilities.

Incompatible, externally-visible upgrades are rare for web services that use HTTP but are more common in non-web services like persistent online games. In such systems, clients are forced to disconnect while the service upgrades and, when they reconnect, are forced to upgrade their client software to the latest version. This ensures that the service never needs to support old behaviors and that all clients see the same version of the service. Some systems support such upgrades by implementing multiple versions. For example, NFS servers implement both NFSv2 and NFSv3. The problem with this approach is that there is no barrier between these implementations, so one can corrupt the other; simulation objects prevent this by modularizing the implementation, and they furthermore make it easy to retire the old code.

Our methodology supports all these kinds of upgrades and enables systems to provide service during incompatible upgrades via simulation. Simulation eliminates the need to take down whole datacenters for incompatible upgrades and can allow clients to delay upgrading until convenient. Our methodology is especially important for peer-to-peer systems, since in those systems there are no tiers or clients; rather every node must upgrade, the upgrade must happen gradually, and even compatible upgrades require simulation so that upgraded nodes can call new methods on non-upgraded nodes.

10 Conclusions

We have presented a new automatic upgrade system. Our approach targets upgrades for large-scale, long-lived distributed systems that manage persistent state and need to provide continuous service. We support very general upgrades: the new version of the system may be incompatible with the old. Such incompatible upgrades, while infrequent, are important for controlling software complexity and bloat. We allow upgrades to be deployed automatically, but under control: upgraders can define flexible upgrade scheduling policies. Furthermore, our system supports mixed mode operation in which nodes running different versions can nevertheless interoperate.

In addition, we have defined a methodology for upgrades that takes mixed mode operation into account. Our methodology defines requirements for upgrades in systems running in mixed mode and provides a way to specify upgrades that enables reasoning about whether the requirements are satisfied. Our specification techniques are modular: only the old and new types of the upgrade must be considered, and possibly the specification of the previous upgrade.

We also presented a powerful implementation approach (running SOs as interceptors) that allows all behavior permitted by the upgrade specification to be implemented. Our approach allows the upgrader to define how long legacy behavior must be supported, by defining the deployment schedule for the incompatible upgrade.

We have implemented a prototype infrastructure called Upstart and shown that it imposes modest overhead. We have also evaluated the usability of our system by implementing a number of examples. The most challenging problem is defining SOs, but they can mostly be implemented by a combination of delegation and use of code that will be in the new version provided by the upgrade.

References

1. Sameer Ajmani. *Automatic Software Upgrades for Distributed Systems*. Ph.D., MIT, September 2004. Also available as technical report MIT-LCS-TR-1012.
2. Joao Paulo A. Almeida, Maarten Wegdam, Marten van Sinderen, and Lambert Nieuwenhuis. Transparent dynamic reconfiguration for CORBA, 2001.
3. S. Amer-Yahia, P. Breche, and C. Souza. Object views and updates. In *Journes Bases de Donnes Avances*, 1996.
4. C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *Intl. Conf. on Configurable Dist. Systems*, pages 35–42, May 1998.
5. Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, 1983.

6. Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA*, 2003.
7. Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, July 2001.
8. B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.
9. Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP*, October 2001.
10. R. S. Fabry. How to design systems in which modules can be changed on the fly. In *Intl. Conf. on Software Engineering*, 1976.
11. Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *NSDI*, San Francisco, CA, March 2004.
12. Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, pages 111–128, 1991.
13. Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Programming Language Design and Implementation*, pages 13–23, 2001.
14. 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.
15. Michael Kaminsky, George Savvides, David Mazières, and M. Frans Kaashoek. Decentralized user authentication in a global file system. In *SOSP*, pages 60–73, October 2003.
16. Deepak Kapur. Towards a theory for abstract data types. Technical Report MIT-LCS-TR-237, MIT, June 1980.
17. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.
18. Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing persistent objects in distributed systems. In *European Conf. on Object-Oriented Programming*, June 1999.
19. Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
20. Simon Monk and Ian Sommerville. A model for versioning of classes in object-oriented databases. In *British National Conf. on Databases*, pages 42–58, Aberdeen, 1992.
21. L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *HotNets I*, October 2002.
22. Tobias Ritzau and Jesper Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, May 2000.
23. Jon Salz, Alex C. Snoeren, and Hari Balakrishnan. TESLA: A transparent, extensible session-layer architecture for end-to-end network services. In *USITS*, 2003.
24. Twittie Senivongse. Enabling flexible cross-version interoperability for distributed services. In *Distributed Objects and Applications*, 1999.
25. Andrea H. Skarra and Staney B. Zdonik. The management of changing types in an object-oriented database. In *OOPSLA*, pages 483–495, 1986.
26. 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 *USENIX Annual Technical Conf.*, 2003.
27. R. Srinivasan. RPC: Remote procedure call specification version 2. RFC 1831, Network Working Group, 1995.
28. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and flexible dynamic software updating. In *Principles of Programming Languages*, 2005.
29. L. A. Tewksbury, L. E. Moser, and P. M. Melliar-Smith. Live upgrades of CORBA applications using object replication. In *ICSM*, pages 488–497, November 2001.