Automatic Software Upgrades for Distributed Systems

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 generic *automatic upgrade system* that enables distributed systems to provide service during upgrades.

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.

Second, our approach allows mixed mode operation. But this raises a 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, and Section 9 discusses the usability of our approach. Section 10 discusses related work, and Section 11 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 [26]); 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 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.

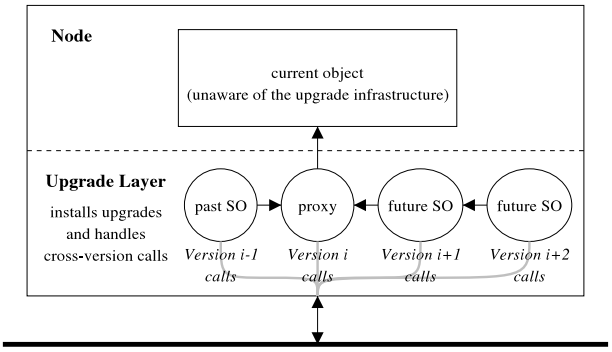

Figure 1: *The structure of a node.*

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, 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 in-

stalls 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 upgrade affects it, i.e., whether the upgrade contains an old class that matches its current class if it isn't running any future SOs, or else the class of its latest future SO. (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 and starts a future SO if necessary, e.g., if the new type is a subtype of the old one.

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. 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 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 understand what it means.

Because of simulation, at a given time, a node may 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 object, while others interact with a simulation object. Yet all these objects 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, 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.

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. Our approach is modular: all the upgrader needs to understand is the old and new types for that upgrade.

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 4.3.

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 when the node upgrades.

Together, the type and sequence requirements can be overconstraining: it may not be possible to satisfy them both 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. 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.

3.1 Specifying Upgrades

Now we describe how to specify an upgrade involving two unrelated types 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. 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 default 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, $T_{new}.shadowT_{old}\$m$ (read 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 $T_{new}.p$.

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:

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

Also, shadow methods must *preserve the invariant*:

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

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:

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

A stronger invariant maps colors to flavors:

$$\begin{aligned} \langle x, \text{blue}\rangle \in O_{old} &\Leftrightarrow \langle x, \text{grape}\rangle \in O_{new}, \\ \langle x, \text{red}\rangle \in O_{old} &\Leftrightarrow \langle x, \text{cherry}\rangle \in O_{new}, \\ &\ldots \end{aligned} \quad (2)$$

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}:

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

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

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

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.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: $this_{post} = this_{pre} - \{\langle x, c\rangle\}$

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: $this_{post} = this_{pre} - \{\langle x, f\rangle\}$

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.shadowColorSet$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 because preserving the invariant I between two objects causes one of them to violate its specification.

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.

The methodology for disallowing is as following:

- Provide shadows for all mutators of the old and new types.
- The future SO must disallow any methods that would cause violations of the specification of the old type. Also if shadows of any old type methods violate the specification of the new type, the future SO must disallow methods that expose these violations.
- The past SO must disallow any methods that would cause violations of the specification of the new type. Also, if any shadows of the new type methods violate the specification of the old type, the past SO must disallow methods that expose these violations.

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.

4 Simulation Objects

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.

4.1 Interceptor 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 future SO. That SO handles all the calls, but it delegates to the preceding object, which may be the current object or another SO.

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. 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. Thus this approach has no past SOs.

The interceptor approach is simple and powerful, because a single object manages all the types of the node. The approach works very well when the new type is a subtype of the old one, because the new object is already prepared to handle all calls to its predecessor; thus no additional effort is required of the upgrader in this case. The approach doesn't work so well for incompatible upgrades nor for upgrades where the the new type is a supertype of the old one. The main problem is that the approach forces continued support for legacy behavior. Supporting legacy behavior for a short time, e.g., just in the future SO for an incompatible upgrade, is acceptable. But using interceptors for a sequence of incompatible upgrades means the upgrader must understand every type that the node supports and the relationships between them. Thus the approach can be highly non-modular, and in this case, the likelihood that the interceptor code is correct declines.

4.2 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).

The direct approach is modular because the upgrader only needs to understand the old and new types of a class upgrade; all earlier upgrades can be ignored, unlike in the interceptor approach. But it 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 seems 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 above 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).

4.3 Concurrency Control

So far we have assumed a node runs calls one-at-a-time. But, a node may process many calls concurrently, and if

the UL imposes restrictions on concurrency, this could lead to reduced performance or even deadlock. Therefore we have no choice but to let the objects on a node—the current object and simulation objects—implement synchronization themselves.

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. But non-interceptors cannot 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. If the SO is an interceptor, it can implement deq2 by calling deq twice on the delegate and ensuring no other deq calls are in progress. But a non-interceptor cannot do this, because a client could call deq on the current object in between the non-interceptor's deq calls.

In a case like this, 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 non-interceptor can use lockdeq to implement deq2 correctly. If the delegate does not provide appropriate concurrency control methods, the upgrader's only choice is to disallow deq2.

4.4 Revisiting Interceptors

The interceptor approach has greater expressive power than the direct approach: it provides full control over interleaving and concurrency and so avoids the need to disallow calls for these reasons. However, the interceptor approach does not work well for incompatible upgrades, as it requires the future SO to support legacy behavior.

However, in practice incompatible upgrades occur less often than compatible ones. And if there is never more than one incompatible upgrade in progress, we can provide an approach in which *all* SOs run as interceptors.

To get an all-interceptor approach, we must consider both future SOs and past SOs.

When an incompatible upgrade is introduced, its future SO can run as an interceptor assuming there is no other incompatible upgrade in progress. Note that the upgrader can easily define such an SO, since he knows about both the old type (T_{old}) and new type (T_{new}).

But now one of two things can happen: another upgrade is introduced, or the incompatible upgrade is installed. Consider first what happens when the next upgrade is introduced, and let's assume that this is a compatible upgrade, since incompatible upgrades are likely to be rare; thus T_{newnew} is a subtype of T_{new}. We can run the future SO for T_{newnew} as an interceptor provided it handles both its own type (T_{newnew}) and the old type of the last incompatible upgrade (T_{old}). Note that having the future SO handle this extra behavior is not much of a burden either for its specification or its implementation: it simply inherits shadow method specifications from T_{new}, and it can delegate calls on T_{old} methods to the T_{new} object.

Now let's consider what happens when the incompatible upgrade is installed. At this point, a past SO must be provided for the old type, and to avoid having it disallow calls, we would like to run it as an interceptor. This means that it must implement both its own type (T_{old}) and the new type (T_{new}), though it can implement the latter by delegating to the current object.

In this example, when a call intended for the current object arrives, it goes first to the future SO for T_{newnew}, then to the past SO for T_{old}, and finally to the current object. Thus the future SO needs to continue to support T_{old} even though the incompatible upgrade has been installed.

This situation continues until the incompatible upgrade is retired. At this point the past SO can be removed; the future SOs will stop receiving any T_{old} calls since this type is no longer in use; and if a new incompatible upgrade is introduced, its future SO can run as an interceptor.

Thus we can run entirely in interceptor mode, thereby allowing full expressibility for SOs, provided incompatible upgrades aren't introduced too frequently. In particular, the previous incompatible upgrade must be retired before the next one is introduced. We believe this is likely to be the common situation in practice.

To write SOs in this all-interceptor approach, the upgrader needs to be aware of the most recent incompatible upgrade, though this may not be the immediately preceding version. However, we still satisfy our modularity constraint, since there is a bound on what the upgrader must know: to implement a new compatible upgrade, the upgrader needs to know the old type of the most recent incompatible upgrade, plus the old and new types of the current upgrade.

If a second incompatible upgrade arrives too early we can run the SOs for the new one using the direct approach (with disallowing) until the previous incompatible upgrade retires, and we might speed up the scheduling of the previous incompatible upgrade to retire it quickly. The alternative would be to incorporate still more knowledge (i.e., the upgrader must know the old types of multiple incompatible upgrades), which is probably not desirable.

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. If a node needs to recover 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 if nodes upgrading simultaneously attempt to obtain state from each other. It also makes TFs simpler to implement and reason about. This restriction does not limit the expressive power of TFs: a transformation is simply a node restart during which the node's persistent representation may change in arbitrary ways.

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. They can also consult additional information: a central database that records the upgrade status of every node and can contain user-defined tables, and per-node local databases that record information about the status of other nodes with which this 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. How this is accomplished depends on how the system is designed. Consider the upgrade schedules described in [9]: 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. A *big flip* causes half the nodes in a system to upgrade at once; this makes sense for systems that need to upgrade quickly during off-peak hours 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. The implementations of these SFs are each just a few lines of script.

By combining per-node and centralized information, a variety of other schedules are possible, such as "wait until the node's servers upgrade," "wait until all nodes of class C upgrade," "wait for the all clear signal," "wait until the node is lightly loaded," and "avoid creating blind spots in the sensor network." A schedule can be chosen to avoid implementing difficult SO features; the first schedule above might be chosen for this reason.

7 Example

We present a brief example to illustrate our approach. Our upgrade replaces a replicated file server that uses Unix-style permissions with one that uses per-file access control lists (ACLs) [17]. It also switches clients from using permissions to using ACLs. Thus, this upgrade contains two class upgrades: one for clients and one for servers.

Scheduling is simple: the client SF waits until the client is idle. The server SF upgrades replicas round-robin.

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: ⟨*filename, content, owner, or, ow, ox, group, gr, gw, gx, wr, ww, wx*⟩. Only the owner of a file can modify the file's permissions, group, or owner. The new state (O_{new}) is a set of ⟨*filename, content, acl*⟩ tuples, where *acl* is a sequence of zero or more ⟨*principal, r, w, x, a*⟩ tuples. Principals with the *a* permission are allowed to modify the ACL.

Now we can define an invariant $I(O_{old}, O_{new})$:

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

This 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 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 the default permissions differently. The invariant above is a particularly weak (i.e., permissive) one.

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, then clients of the permissions system may see modifications made by clients of the new system that do not appear to have the correct permissions. To prevent this, we might disallow such operations in the future SO, but of course we cannot disallow them once the server has upgraded; at this point, users of the permissions system will observe odd behavior, unless we refuse to allow them to do anything at all. For example, an owner in the ACL system might add as a second owner a client of the permissions system, and then later remove that client as an owner.

Clearly we don't want to prevent users of the permissions system access to files. Furthermore, 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 our solution is to allow all methods in both the past and future SO. The shadow methods spell out the details; for example, a modification to the ACL may cause the corresponding permissions to change.

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 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). We discuss an implementation of this TF in Section 9.2.

While this example seems simple initially, on closer inspection it has several subtleties. Such subtleties are common in real systems; the purpose of our methodology is to help upgraders design correct upgrades nonetheless.

8 Evaluation

This section evaluates Upstart, our upgrade infrastructure.

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 [13] to cache and serve the code.

The upgrade database (UDB) is implemented 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 they will be resent later.

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.

To minimize communication overhead, we implement 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 [26]. Providing support for other kinds of systems is straightforward and requires no changes to the upgrade infrastructure.

8.1 Experiments

The most important performance issue is the overhead imposed by the upgrade layer, both when no upgrades are happening, and when running simulation objects. This section presents experiments that measure these overheads and show them to be reasonable. 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. Finally, *With SO* measures the performance of the benchmark with a null SO on the server, i.e., an SO that just delegates; it adds the overhead of dispatching calls through the SO, unmarshaling data from the network into RPCs, and remarshaling RPCs to pass them to the benchmark.

Figure 2 shows the cumulative distribution function (CDF) for null RPC latencies over the loopback interface. In this experiment, a client issues empty RPCs to a server one-at-a-time using UDP. The *Baseline* median latency is 51μs; *TESLA*, 139μs; *Upstart*, 206μs; and *With SO*, 215μs. 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. The extended tail for *With SO* comes from delays due to downloading and installing the SO.

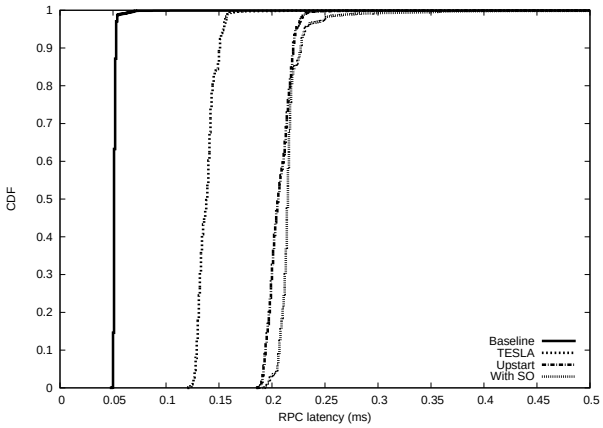

Figure 2: *Null RPCs over the loopback interface (N=100000)*

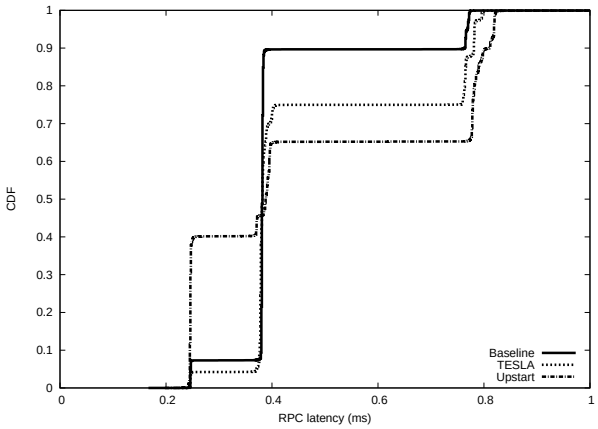

Figure 3: *Null RPCs over a gigabit ethernet crossover cable (N=100000)*

Figure 3 shows the CDFs for the latencies of null RPCs over a crossover cable. The *Baseline* median latency is 382μs; *TESLA*, 382μs; *Upstart*, 388μs. We omit *With SO*, as it provides no additional information in this case. The CDFs stair-step at 125μs intervals; we believe 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. We believe the reason 40% of Upstart's messages have lower

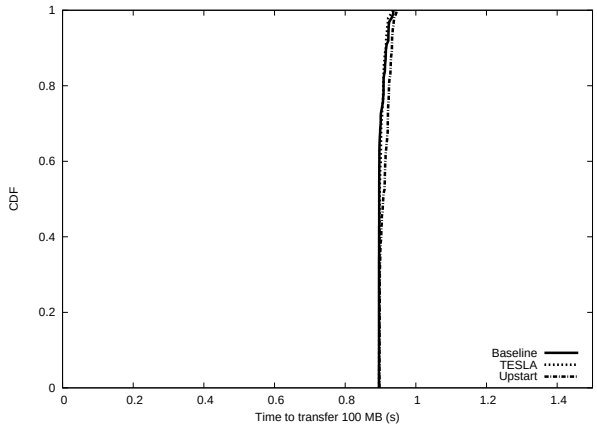

Figure 4: *100 MB TCP transfer over a gigabit ethernet crossover cable (N=100)*

latency than Baseline is because of the difference in Upstart's messages' processing time and size.

Figure 4 shows latencies for an experiment in which a client transfers 100 MB of data to a server using TCP (without RPCs) over a crossover cable. The median *Baseline* throughput is 893 MB/s; *TESLA*, 891 MB/s; and *Upstart*, 882 MB/s. 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.

9 Usability

This section evaluates the usability of our approach by considering how hard it is to implement upgrades. We begin by discussing SOs, since this is where the upgrader is likely to do the most work. Then we discuss our experience implementing upgrades.

9.1 Simulation Objects

This section discusses implementing SOs that run as interceptors, since this is the hardest and also the most likely case (as discussed in Section 4.4).

Let's consider compatible upgrades first. In this case the future SO can delegate supertype methods calls to its predecessor. But it may still need to do quite a bit of work itself because sometimes the behavior added by the new type is complex. For example, consider a document-serving system and an upgrade that adds the ability to translate documents into new languages. The new version contains code to implement translation and to keep a persistent cache of translated documents as an optimization.

The future SO for this upgrade must also provide document translation. However, since it is implemented in conjunction with the new version, it can do translation by using the new translation library. But how to store translations persistently? In this example, the best option is probably to retain only a volatile cache. But if the new state had to be persistent, one option is to use a simple but perhaps inefficient approach, e.g., just log new information in a file or a persistent store like BerkeleyDB [20]. The other (more efficient) option is to provide persistence by using code provided in the new current object for this purpose. Thus providing persistence isn't difficult.

Of course, if the future SO has persistent storage, this will affect the TF. The upgrader needs to keep this in mind, and choose a persistent storage approach that is both easy for the SO to use, and easy for the TF to interpret.

Now let's consider incompatible SOs. A future SO (or past SO) for an incompatible upgrade has to do two things: implement the new behavior, and implement the shadow methods. The former can be done exactly as discussed above, but implementing shadow methods is additional work. For example, consider the permissions to ACLs example discussed in Section 7, and suppose the future SO for ACLs allows the setAcl method. Then it will need to compute the effect of a setPerms call on the ACL, and it will need to compute the effect of a setAcl call on permissions and call the appropriate methods of the old object to effect the proper modifications.

The main point is the following: The shadow methods are the one place where the upgrader has extra work to do, i.e., where the work can't be done mostly by delegation. And if shadow methods are hard to implement, the right thing may be for the upgrader to disallow behavior; disallowing methods to reduce the difficulty of implementing an SO is always acceptable.

9.2 Experience

This section reports on our experience with upgrades.

Simulation Objects. We evaluated the difficulty of writing simulation objects with an upgrade for a small application called DocServer. This example is interesting because there is new behavior, new persistent state, and it is an incompatible upgrade. DocServer stores a persistent mapping from names to documents, and makes the mapping persistent by keeping it in a local database. The upgrade allows clients to attach persistent comments to documents. This upgrade is incompatible because the new type no longer supports the getDoc method (the new type only has getDocWithComments).

Since the new version introduces persistent comments, the future SO must maintain this state. Our implementation keeps the state in a local database. The past SO is trivial: it has no persistent state, and it implements getDoc by calling getDocWithComments and stripping out the comments.

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

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.

Transform functions. To evaluate the difficulty of writing transform functions, we implemented a TF that adds an access control list to every file and directory in a file system, as required by our example upgrade in Section 7. This upgrade is interesting because it needs to transform large amounts of state.

The ACL format is that of SFSACL [17]: 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 or directory's ACL are derived from the Unix permissions of that file or directory. The TF assumes no additional ACL state is kept by the future SO; if there were such state, the TF would need to obtain ACLs from it instead.

An 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.

Scheduling functions. Scheduling functions are easy to implement: they are typically small scripts that wait for a particular time or for some query in the upgrade database to be satisfied. Furthermore, it would be easy to provide a library of common SFs so that usually upgraders won't need to implement their own.

Of course, we would like to know whether upgrade schedules work in practice, especially in large systems. To answer this question, we defined and ran a simple upgrade on PlanetLab, a large research testbed [21]. Specifically, we deployed DHash [11], a peer-to-peer storage system, on 205 nodes and installed a null upgrade on it.

Defining the upgrade was straightforward: no TF or SOs were required. The SF upgraded 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.[1]

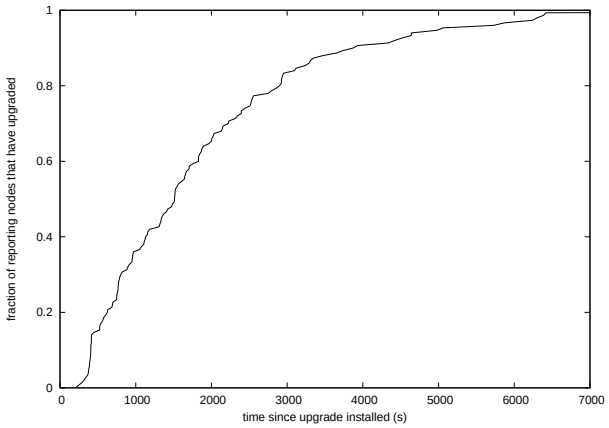

Figure 5: *Cumulative fraction of upgraded nodes on PlanetLab.*

The upgrade ran as expected, and the DHash network remained functional throughout. Figure 5 depicts the fraction of nodes upgraded over time. This experiment identified the need to have nodes retransmit their status periodically, as the UDB never received the post-upgrade status update for 54 of the nodes! (These are omitted from the graph.) In the future, we plan to re-run these experiments to get the full upgrade trace.

We also ran an experiment to evaluate the effect of an upgrade on DHash client performance. Here the system consisted 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 6 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 6 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. In the future, we plan to run similar experiments with larger networks.

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

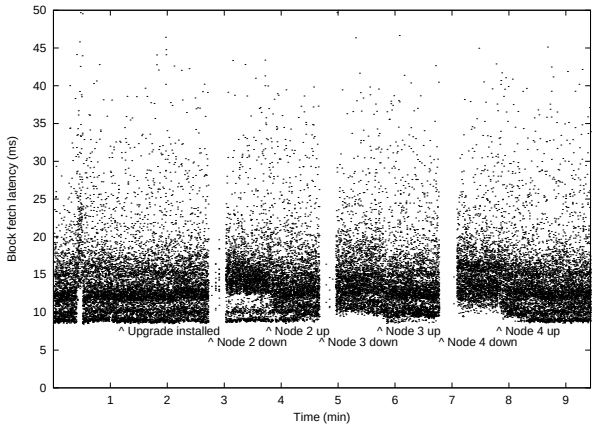

Figure 6: *DHash block fetch performance during an upgrade.*

10 Related Work

In earlier work, distributed upgrades were studied in systems with a wide variety of requirements, some similar, some different from ours. We consider these requirements and the techniques developed in the earlier systems.

The traditional approach in systems that can tolerate downtime is to synchronize the software on a set of nodes using package installers or centralized software management systems; we do not discuss these further here.

Reconfigurable distributed systems [4, 6, 7, 16, 18, 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 stall when nodes in the subsystem fail.

Systems that must provide continuous service during upgrades sometimes use custom schedules [9]. Google [10] upgrades entire data centers at once and provides service by redirecting clients to other data centers. This approach only supports protocol changes within data centers. Gnucleus [2], a peer-to-peer file-sharing system, disseminates upgrades eagerly by gossip. There is no support for interaction between versions, so service may be interrupted for incompatible upgrades.

Many systems support upgrades by requiring backwards compatibility. These systems are tiered, and nodes in higher tiers upgrade before those in lower tiers; for example, storage servers upgrade before application servers, which in turn upgrade before front-end servers. Rolling upgrades [9] are used to upgrade nodes within tiers.

Some systems support incompatible upgrades with implementations that handle multiple versions at once, e.g., NFS servers implement both NFSv2 and NFSv3. There is no barrier between these implementations, so one can corrupt the other; SOs prevent this.

Many systems, e.g., Google and Gnutella [3], support limited kinds of incompatible upgrades using *extensible protocols*, consisting of a baseline protocol and a set of extensions that only some nodes handle. The problem here is that nodes must function correctly with or without the extensions, which complicates software design.

A few systems support cross-version interaction using wrappers: PODUS [14] supports asynchronous upgrades to individual procedures in a (possibly distributed) program, and The Eternal system [27] supports asynchronous 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 [8], others [19, 25] use mixed mode: they allow objects of different versions to interact. The work on views in O2 [5] 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* [12, 15] is to enable a node to upgrade its code and transform its volatile state without shutting down. These techniques require the implementor to identify where in the program reconfiguration can take place and are typically language-specific. In contrast, our system uses a language-independent approach and only preserves persistent state. Nonetheless, these approaches are complementary to ours and could be used to reduce downtime during upgrades.

11 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 versions of the upgrade must be considered.

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 here is also modular, although we extended what the upgrader needed to know to include in addition the old version of the previous unretired incompatible upgrade. 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] Anonymized.

[2] The Gnucleus open-source Gnutella client. `http://www.gnucleus.com/Gnucleus/`.

[3] The Gnutella file sharing protocol, 2000. `http://rfc-gnutella.sourceforge.net`.

[4] Joao Paulo A. Almeida, Maarten Wegdam, Marten van Sinderen, and Lambert Nieuwenhuis. Transparent dynamic reconfiguration for CORBA, 2001.

[5] S. Amer-Yahia, P. Breche, and C. Souza. Object views and updates. In *Journes Bases de Donnes Avances*, 1996.

[6] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *ICCDS*, pages 35–42, Annapolis, MD, May 1998.

[7] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, 1983.

[8] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA*, 2003.

[9] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, July 2001.

[10] 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.

[11] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP*, October 2001.

[12] R. S. Fabry. How to design systems in which modules can be changed on the fly. In *ICSE*, 1976.

[13] Michael J. Freedman, Eric Freudenthal, and David Mazires. Democratizing content publication with Coral. In *NSDI*, San Francisco, CA, March 2004.

[14] 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.

[15] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *PLDI*, pages 13–23, 2001.

[16] 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.

[17] 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.

[18] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[19] Simon Monk and Ian Sommerville. A model for versioning of classes in object-oriented databases. In *BNCOD 10*, pages 42–58, Aberdeen, 1992.

[20] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *USENIX*, 1999.

[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 *DOA*, 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] R. Srinivasan. RPC: Remote procedure call specification version 2. RFC 1831, Network Working Group, 1995.

[27] 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.