Automatic Software Upgrades for Distributed Systems

Sameer Ajmani
ajmani@lcs.mit.edu

April 30, 2003

Abstract

Upgrading the software of long-lived distributed systems is difficult. It is not possible to upgrade all the nodes in a system at once, since some nodes may be down and halting the system for an upgrade is unacceptable. This means that different nodes may be running different software versions and yet need to communicate, even though those versions may not be fully compatible. We present a methodology and infrastructure that addresses these challenges and makes it possible to upgrade distributed systems automatically while limiting service disruption.

1 Introduction

Our goal is to support automatic software upgrades for long-lived distributed systems, like server clusters, content distribution networks, peer-to-peer systems, and sensor networks. Since such systems are long-lived, we must expect changes (upgrades) in their software over time to fix bugs, add features, and improve performance. The key challenge to upgrading such systems is enabling them to provide service during upgrades.

Ideally, a distributed system would upgrade the software on all its nodes simultaneously and instantaneously and then would resume operation from the "appropriate" state. In reality, upgrades are not instantaneous: they take time, so nodes will be unavailable while they upgrade.[1] A good upgrade is one that upgrades all the nodes in the system in a timely manner, transforms their persistent state from the old version's representation to one that makes sense in the new version, and does so with minimal service disruption.

Earlier approaches to automatically upgrading distributed systems [11–13, 25, 27, 35] or distributing software over networks [1–5, 7, 21, 39, 42] do little to ensure continuous service during upgrades. The Eternal system [41], the Simplex architecture [38], and Google [19] enable specific kinds of systems to provide service during upgrades, but they do not provide general solutions.

Our approach to upgrading takes advantage of the fact that long-lived systems are *robust*: they tolerate node failures, and they allow nodes to recover and rejoin the system. Nodes are prepared for failures and know how to recover to a consistent state. This means that we

[1] Dynamic updating systems [17, 18, 20, 23, 24, 30] can reduce the time required to upgrade a node, but they still require time to install new code and to transform a node's state.

can model a node upgrade as a soft restart. But even with this assumption, there are several challenges that a general approach for upgrading distributed systems must address:

First, the approach must provide a way to define upgrades, propagate these definitions to nodes, and cause the upgrade to happen, while allowing administrators to monitor and control upgrade progress. For large-scale or embedded systems, it is not practical for a human to control upgrades, e.g., by doing a remote login to a node from some centralized management console and installing the upgrade, taking care of any problems as they arise. Therefore, upgrades must propagate automatically.

Second, the approach must enable the system designer to control *when* individual nodes upgrade. It is not practical to halt the system and then cause all nodes to upgrade, since at any moment some nodes may be powered down or not communicating. Furthermore, providing continuous service may require that just a few nodes upgrade at a time. Also, the designer may want to test the upgrade on a few nodes to verify that the new code is working satisfactorily before upgrading other nodes.

Third, since nodes upgrade at different times, the approach must enable the system to provide service when nodes are running different versions. This is simple when upgrades just correct errors or extend interfaces. But upgrades might also change interfaces in *incompatible* ways: a node might cease supporting some behavior that it used to, either because providing the old behavior would be uneconomical or because the upgraded system provides the behavior in a different way. These incompatible changes are the most difficult to deal with because they can lead to long periods when nodes that need to communicate assume different interfaces. A system must continue to run (possibly in a somewhat degraded mode) even when such incompatibilities exist.

Finally, the approach must provide a way to transform the persistent state of nodes from the old versions' representation to that of the new version. These transformations must capture the effects of all previous operations made on the nodes, since clients depend on the upgraded system to maintain that information.

To address these challenges, our approach includes an an *upgrade infrastructure*, *scheduling functions*, *simulation objects*, and *transform functions*.

The *upgrade infrastructure* is a combination of centralized and distributed components that enables rapid dissemination of upgrade information and flexible monitoring and control of upgrade progress. This thesis will describe a prototype of the infrastructure and will evaluate it in several scenarios.

Scheduling functions (SFs) are procedures that run on nodes and tell them when to upgrade. Scheduling functions make it simple to define upgrade schedules like "upgrade server replicas one-at-a-time" and "upgrade a client only after its servers upgrade." A designer can customize an upgrade schedule to the capabilities and needs of the system being upgraded. But this generality has risks: a bad upgrade schedule could disrupt service or could cause an upgrade to stall. This thesis will define a framework to support scheduling functions and will investigate techniques for mitigating their risks.

Simulation objects (SOs) are adapters that allow a node to behave as though it were running multiple versions simultaneously. Unlike previous approaches that propose similar adapters [18,32,36,40,41], ours includes correctness criteria to ensure that simulation objects reflect node state consistently across different versions. These criteria require that some in-

teractions made via SOs must fail; we identify when such failure is necessary and, conversely, when it is possible to provide service between nodes running different versions.

Transform functions (TFs) are procedures that convert a node's persistent state from one version to the next. This thesis will explain how TFs interact with SOs to ensure that nodes upgrade to the correct new state. This thesis will not attempt to automate the generation of transform functions or simulation objects, although previous work suggests that this is possible [23, 28, 41].

The rest of the proposal is organized as follows. Section 2 presents our upgrade model and assumptions. Section 3 describes the upgrade infrastructure, and Section 4 introduces an upgrade example. Section 5 describes scheduling functions; Section 6, simulation objects; and Section 7, transform functions. Section 8 presents informal correctness criteria for simulation objects and transform functions. Section 9 discusses related work. Section 10 gives a schedule for the thesis, Section 11 identifies required facilities, and Section 12 discusses future work.

2 Model and Assumptions

We model each node as an object that has an identity, a type, and state. A node is fully-encapsulated, so the only way of accessing its state is by calling its methods. A node runs some top-level class that implements its type. The underlying implementation may be a large code image composed of many components and libraries, but we only consider a node's top-level behavior. Systems based on remote procedure calls [**?**] or remote method invocations [**?**] map easily to this model. Extending this model to message-passing [**?**] is future work. We also assume there is just one object per node; supporting multiple objects per node is also future work.

A portion of an object's state may be persistent. Objects are prepared for failure of their node: when the node recovers, the object reinitializes itself from the persistent portion of its state.

Our approach defines upgrades for entire systems, rather than just for individual nodes. A *version* defines the software for *all* the nodes in the system (like a schema for an object-oriented database defines the types for all the objects in the database). Different nodes may run different top-level classes, e.g., a version can define one class for clients and another for servers. We say that a node is running *at* a particular version if it is running some class defined by that version. If two nodes are running at the same version, then we assume that they can communicate correctly with one another, e.g., a client and a server running at the same version should use the same communication protocol.

An upgrade moves the system from one version to the next. We expect upgrades to be relatively rare, e.g., they occur less than once a month. Therefore, the common case is when all nodes are running the same version. We also expect that before an upgrade is installed, it is thoroughly debugged; our system is not intended to providing a debugging infrastructure.

An upgrade identifies the classes that need to change by providing a set of *class upgrades*: ⟨*old-class, new-class, TF, SF, past-SO, future-SO*⟩. *Old-class* identifies the class that is now obsolete; *new-class* identifies the class that is to replace it. TF is a *transform function* that generates the new object's persistent state from that of the old object. SF is a *scheduling function* that tells a node when it should upgrade. *Past-SO* and *Future-SO*

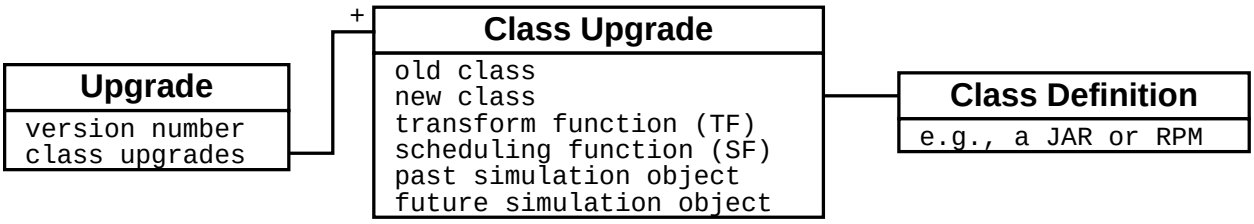


Figure 1: The structure of an upgrade definition.

are classes providing *simulation objects*. *Past-SO*'s object implements old-class's behavior by calling methods on the new object (i.e., it provides backward compatibility); *Future-SO*'s object implements new-class's behavior by calling methods on the old object (i.e., it provides forward compatibility). An important feature of our approach is that the upgrade designer only needs to understand the new version and the old one.

We assume class definitions are stored in well-known repositories and define the full implementation of a node, from its top-level behavior down to its libraries and operating system. Package systems [1, 5, 7] provide standard ways of encapsulating class definitions. Figure 1 depicts the full structure of an upgrade definition.

Sometimes new-class will implement a subtype of old-class, but we do not assume this. When the subtype relationship holds, no past-SO is needed, since new-class can handle all calls for old-class. Often, new-class and old-class will implement the same type (e.g., when new-class just fixes a bug or optimizes performance), in which case neither a past-SO nor a future-SO is needed.

We assume that all nodes running the old-class must switch to the new-class. Eventually we may provide a filter that restricts a class upgrade to only some nodes belonging to the old-class; this is useful, e.g., to upgrade nodes selectively to optimize for environment or hardware capabilities.

3 Infrastructure

The upgrade infrastructure consists of four kinds of components, as illustrated in Figure 2: an *upgrade server*, an *upgrade database*, and per-node *upgrade layers* and *upgrade managers*.

A logically centralized *upgrade server* maintains a *version number* that counts how many upgrades have been installed in the past. An upgrade can only be defined by a trusted party, called the *upgrader*, who must have the right credentials to install upgrades at the upgrade server. When a new upgrade is installed, the upgrade server advances the version number and makes the new upgrade available for download. We can extend this model to allow multiple upgrade servers, each with its own version number.

Each node in the system is running at a particular version, which is the version of the last upgrade installed on that node. A node's *upgrade layer* labels outgoing calls made by its node with the node's version number. The upgrade layer learns about new upgrades by querying the upgrade server and by examining the version numbers of incoming calls.

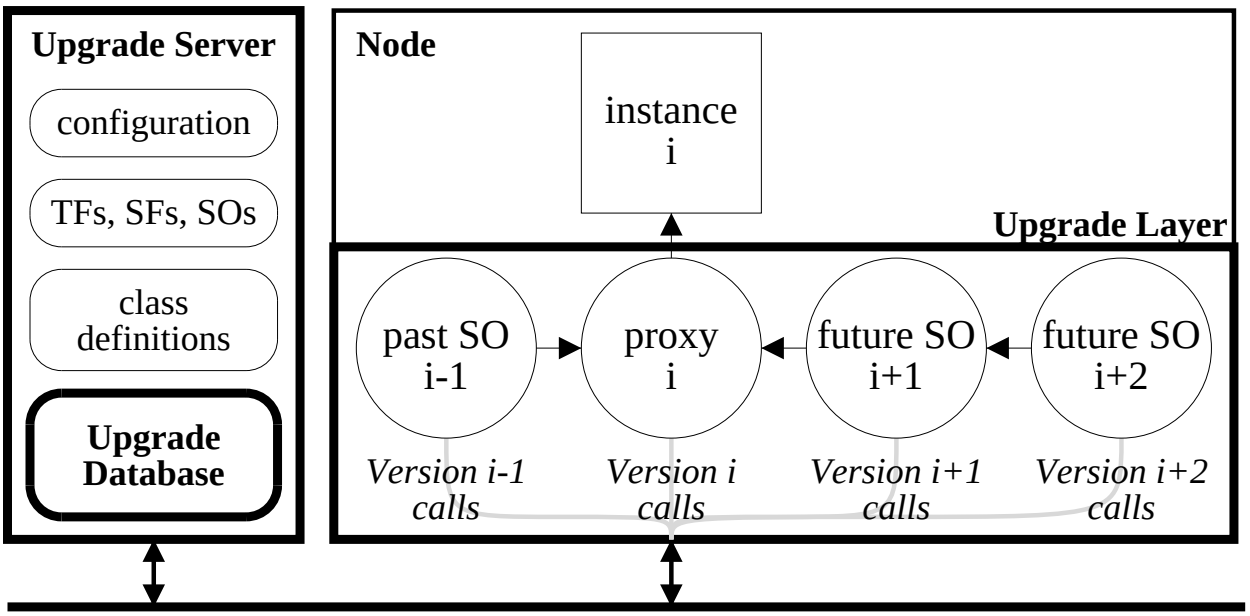


Figure 2: An example system that consists of an upgrade server and a single system node. Arrows indicate the direction of (remote) method calls. The upgrade server stores the upgrade database and publishes upgrades for versions 1 through 4. The node is running at version 2, has a chain of past-SOs for versions 0 and 1, and has a chain of future-SOs for versions 3 and 4.

This exchange of version numbers is an optimization that propagates information about new upgrades in a very lightweight way.

When an upgrade layer hears about a new version, it notifies the node's *upgrade manager*. The upgrade manager downloads the upgrade for the new version from the upgrade server and checks whether the upgrade contains a class upgrade whose old-class matches the node's current class. If so, the node is affected by the upgrade. Otherwise, the node is unaffected and immediately advances its version number.

If a node is affected by an upgrade, its upgrade manager fetches the appropriate class upgrade and class implementation from the upgrade server. The upgrade manager verifies the class upgrade's authenticity then installs the class upgrade's future-SO, which lets the node support (some) calls at the new version. The node's upgrade layer dispatches incoming calls labeled with the new version to the future-SO.

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

In response to the scheduling signal, the upgrade manager shuts down the current node software, causing it to persist its state. The upgrade manager then installs the new class implementation and runs the transform function to convert the node's persistent state to the representation required by new version. The upgrade manager then discards the future-

SO and installs the past-SO, which lets the node continue to support the previous version. Finally, the upgrade manager starts the new version's software, which recovers from the newly-transformed persistent state.

4 Example: a document storage system

This section describes an example system and two upgrades to that system. Later sections will refer back to this example.

The system consists servers that provide document storage and retrieval and clients that interact with those servers. Servers are replicated for high-availability. Clients call methods on servers and retry calls on replicas if calls on the primary fail. In version 1 of our system, servers implement the following interface:

`set(name, string)` Sets the contents of the document named `name` to `string`.

`get(name):string` Returns the contents of the document named `name`, or throws `notfound` if no contents have been set for that document.

Version 2 adds support for comments on documents and changes the semantics of `get` to return a document and its comments together:

`addComment(name, string)` Adds a comment with the value `string` to the end of a list of comments for the document named `name`, or throws `notfound` if no contents have been set for that document.

`set(name, string)` *(unchanged)* Sets the contents of the document named `name` to `string`.

`get(name):string` Returns the concatenation of the contents of the document named `name` with the list of comments for that document, or throws `notfound` if no contents have been set for that document.

This is an incompatible upgrade: version 2 changes the semantics of `get`, so a version 2 server's response to `get` may confuse version 1 clients.

Version 3 fixes a semantic bug: since comments usually depend on the content of a document, the comments for a document should be cleared when the document changes:

`addComment(name, string)` *(unchanged)* Adds a comment with the value `string` to the end of a list of comments for the document named `name`, or throws `notfound` if no contents have been set for that document.

`set(name, string)` Sets the contents of the document named `name` to `string`. Clears the list of comments for that document.

`get(name):string` *(unchanged)* Returns the concatenation of the contents of the document named `name` with the list of comments for that document, or throws `notfound` if no contents have been set for that document.

This is also an incompatible upgrade: version 3 changes the semantics of `set`, so a version 3 server's handling of `set` may confuse version 2 clients.

5 Scheduling Functions

Scheduling functions (SFs) are procedures defined by the upgrader that tell nodes when to upgrade (by *signaling* the nodes' upgrade managers). Unlike existing systems that coordinate upgrades centrally [4,21,39], SFs run on the nodes themselves. This lets SFs respond quickly to changing environments, e.g., to avoid upgrading a replica if another one fails. This approach can also reduce communication and so save energy in resource-constrained systems.

Here are examples of upgrade schedules and SFs:

Upgrade eagerly. The SF signals immediately. This schedule is useful to fix a critical bug.

Upgrade gradually. The SF decides whether to signal by periodically flipping a coin. This schedule can avoid causing too many simultaneous node failures and recoveries, e.g., in a peer-to-peer system.

Upgrade one-replica-at-a-time. The SF signals if its node has the lowest IP address among its non-upgraded replicas. This schedule is useful for replica groups that tolerate only a few failures [6, 41].

Upgrade after my servers upgrade. The SF signals once its node's servers have upgraded. This schedule prevents a client node from calling methods that its servers do not yet fully support.

Upgrade all nodes of class C1 before nodes of class C2. The SF queries the upgrade database to determine when to signal its UM. This schedule imposes a partial order on node upgrades.

Upgrade only nodes 1, 2, and 5. This schedule lets the upgrader test an upgrade on a few nodes [39].

Many other schedules are possible, e.g., to avoid disturbing user activity or to avoid creating blind spots in sensor networks.

In general, we cannot predict what parts of a node's state an SF might use to implement its policy. Instead, we provide SFs with read-only access to *all* of a node's state via privileged observers. Restricting SFs to read-only access prevents them from violating a node's specification by mutating its state.

An SF may also need to know the versions and classes of other nodes. The upgrade database (UDB) provides a generic, central store for such information. Upgrade layers (ULs) store their node's class and version in the UDB after each upgrade. SFs can query the UDB to implement globally-coordinated schedules, and the upgrader can query the UDB to monitor upgrade progress. ULs also exchange this information with other ULs and cache it, so SFs can query ULs for information about recently-contacted nodes. The upgrader can define additional upgrade-specific tables in the UDB, e.g., a list of nodes that are authorized to upgrade. The upgrader can modify these tables to control upgrade progress.

5.1 Correctness Guidelines

The main challenge in designing scheduling functions is ensuring that they behave correctly. Since SFs control the rate at which nodes upgrade, they can affect a system's availability, fault-tolerance, and performance. We do not yet have formal correctness criteria for upgrade schedules, but some intuitive guidelines are:

1. All nodes affected by an upgrade must eventually upgrade. This means that all SFs must eventually signal, so:
   (a) SFs must not depend on node behavior that may be removed by an upgrade, such as the behavior of other nodes. SFs can depend on the current behavior of the node it runs on and on the upgrade infrastructure itself, although the upgrade server and upgrade database may become unavailable due to failures.
   (b) SFs must not deadlock. This means that wait-for schedules must be loop-free. Schedules like "clients wait-for servers" and "replicas wait-for those with lower node IDs" have this property.
2. While meeting the guideline 1, an upgrade should occur with minimal service disruption. This means:
   (a) SFs should not cause nodes that provide redundancy for the same service to upgrade at the same time.
   (b) If node recovery after an upgrade requires state transfer from other nodes, then SFs should limit the number of nodes that upgrade at the same time to avoid causing excessive state transfer when they recover.
   (c) If an upgrade adds a new service, then SFs should cause nodes that provide the service to upgrade before nodes that use it. This is not necessary if non-upgraded nodes can simulate the new service using a future-SO.
   (d) If an upgrade removes an old service, then SFs should cause nodes that use the service to upgrade before nodes that provide it. This is not necessary if upgraded nodes can simulate the old service using a past-SO.
3. While meeting guidelines 1 and 2, an upgrade should occur as quickly as possible.

We plan to formalize these guidelines as correctness criteria and to investigate mechanisms for verifying or enforcing them.

5.2 Examples

Our example system consists of two classes of nodes, clients and servers, and two upgrades, version 2 and version 3. This means we must define a total of four scheduling functions: one per class per version. Version 2 introduces a new feature (adding comments to documents), so according to guideline 2c, we should upgrade servers before clients. Furthermore, according to guideline 2a, we should avoid updating server replicas for the same set of documents at the same time. Guideline 3 implies that clients should upgrade as soon as their servers have. So, a good version 2 SF for servers is "upgrade replicas one-at-a-time," and a good version 2 SF for clients is "upgrade after my servers have upgraded."

Version 3 changes the semantics of an existing method (`set` clears the comments for a document). This change cannot be characterized as "adding" or "removing" an existing feature, and simulation objects cannot mask the semantic change (as discussed in Section 6.3). This suggests that no schedule can satisfy guideline 2 well, so the best SF is one that follows guidelines 1 and 3, i.e., one that upgrades all the nodes as quickly as possible.

6 Simulation Objects

Simulation objects (SOs) enable a system to provide service when it is in *mixed mode*, i.e., when its nodes are running different versions. Mixed mode arises because different nodes upgrade at different times. For a mixed-mode system to provide service, it must be possible for nodes running older versions to call methods on nodes running newer versions, and vice versa. It is important to enable calls in both directions, because otherwise a slow upgrade can partition upgraded nodes from non-upgraded ones (since calls between those nodes will fail).

Several techniques exist for supporting mixed mode; before we detail the design of simulation objects, we discuss the alternatives.

6.1 Supporting Mixed Mode

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

It is important to understand what it means for a single node to implement several versions simultaneously. On one hand, each version has its own specification and so may behave differently than the other versions implemented by that node. On the other hand, all the versions share the identity of a single node. We argue that for different versions to share an identity, they must share *state* appropriately.

Consider the alternative: a node could implement multiple versions by by running instances of each version side-by-side, each with its own state, as depicted in Figure 3(a). For example, a node might implement two versions of a file system specification by running instances of both versions side-by-side. A caller that interacts only with one of the two instances can store and retrieve files as usual. Two callers that interact with the same instance can share files. But if two callers are running at different versions and so interact with different instances, then they cannot share files. A single caller may lose access to its own files, e.g., by storing files at one version, then upgrading, then attempting to fetch files at the next version. Since each instance of the file system has its own state, the files that the caller stored at one version are inaccessible at the next.

To allow multiple versions to share a single identity, the implementations of those versions must share a single copy of the node's state. A straightforward way to do this is to allow them to share state directly, as illustrated in Figure 3(b). A common example of this model is a relational database system that supports multiple views on a single set of tables. The problem with this model is that different versions may have different expectations about what values the state may have (*representation invariants*) and how the state may change over time (*history constraints*). If different versions disagree on these expectations, then they cannot safely share state. One might enforce safe sharing by requiring that each version obey the invariants and constraints of *all* the versions on the node. This has two problems: first,

one would have to re-implement each version each time a new version added an invariant or constraint; second, the state may become so over-constrained that it is useless.

To ensure that a node provides at least some useful functionality, one can designate one version as the *primary* version for the node. The node runs an instance of the primary version's implementation and so can support calls to that version perfectly. To support other versions, the node runs *handlers*.

The simplest handler-based model is illustrated in Figure 3(c) and is similar to Skarra and Zdonik's schema versioning model for OODBs [40]. Calls to the primary version are dispatched to an instance, and calls to other versions are dispatched to stateless error handlers that can substitute results for those calls. This model works only if the error handlers can return a sensible default for the calls they implement. This model is also limited in two ways: first, handlers cannot implement behaviors that use (observe or mutate) the instance's state, and second, handlers cannot implement behaviors that use state outside that of the instance.

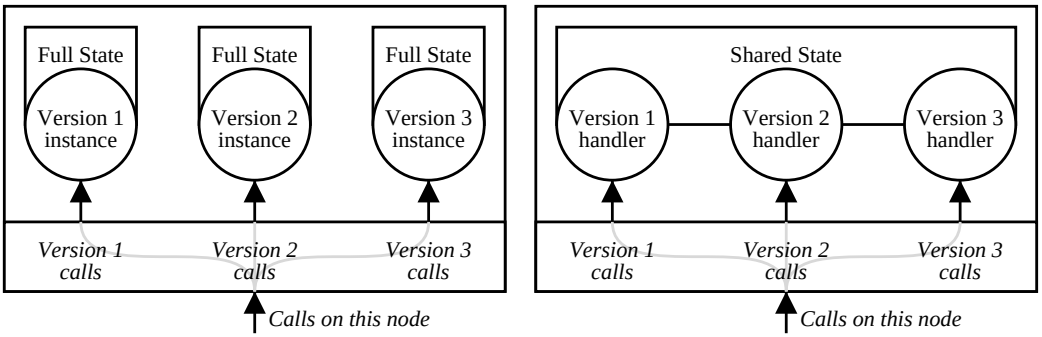
One can address the first limitation by allowing handlers to access the instance's state via its methods, as illustrated in Figure 3(d). This model lets handlers share state safely and lets them support behaviors that can be defined in terms of the instance's behavior. The problem with this model is that to enable a node running any primary version to support calls at any other version, one must define a handler for every pair of versions. This becomes impractical as the number of versions increases.

One can keep the number of handlers manageable using *handler chaining*: each version has just two handlers defined for it, one that calls methods of the next higher version and another that calls methods of the next lower version. Thus, a chain of handlers can map a call on any version to calls on the primary version. Instances of the handler chaining model include Monk and Somerville's update/backdate model for schema versions in OODBs [32] and Senivongse's "evolution transparency" model for distributed services [36].

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

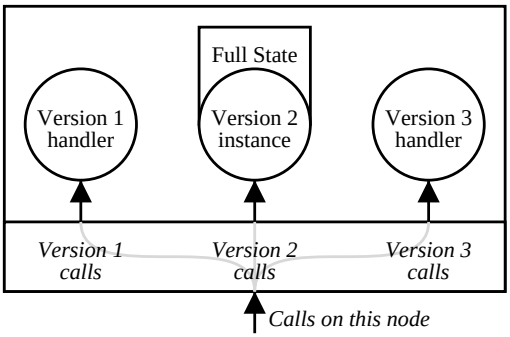
None of these previous models address the second limitation mentioned above: they do not allow handlers to implement stateful behaviors that cannot be defined in terms of the primary instance's behavior. Our solution addresses this limitation by allowing handlers—called *simulation objects*, or *SOs*—to implement calls both by accessing the state of the instance via its methods and by accessing *their own* state, as illustrated in Figure 3(e).

Section 8 defines correctness criteria that restrict the ways in which simulation objects can use their own state to ensure that the states of different versions appear *consistent* with one another. These restrictions may prevent SOs from simulating certain behaviors, so some calls made to SOs may fail. If a new version does not admit good simulation, the upgrader may choose to use an eager upgrade schedule and avoid the use of SOs altogether—but the upgrader must bear in mind that an eager schedule can disrupt service.

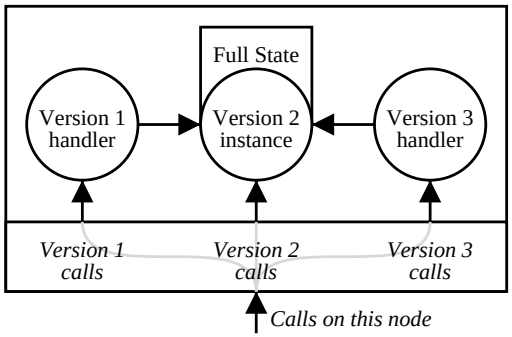


(a) Multiple Instances

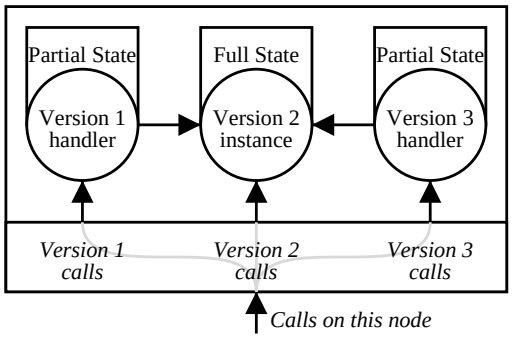
(b) Shared State



(c) Disconnected Handlers



(d) Connected Handlers



(e) Simulation Objects

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

6.2 Simulation Object Mechanics

An upgrader defines two simulation objects for each version, a *past-SO* and a *future-SO*. A *past-SO* implements a previous version's interface by calling methods on the object of the next newer version; thus, a chain of past SOs can support many old versions. It is installed when a node upgrades to a new version and is discarded when the infrastructure determines (by consulting the UDB) that it is no longer needed.[2]

A *future-SO* implements a new version by calling methods on the previous version; like past-SOs, future-SOs can be chained together to support several versions. A future-SO is installed when a node learns of a new version and can be installed "on-the-fly" when a node receives a call at a version newer than its own. A future-SO is removed when its node upgrades to the new version.

At a given time, a node may contain a chain of past-SOs and a chain of future-SOs, as depicted in Figure 2. An SO may call methods on the next object in the chain; it is unaware of whether the next object or is the instance of the primary version or is another SO. An SO may also call methods on other nodes at the SO's own version, however, these calls may fail if the receiving node cannot simulate them.

When a node receives a call, its upgrade layer dispatches the call to the object that implements the version of that call, e.g., the upgrade layer dispatches a `get` call from a version 1 caller to the version 1 object and dispatches a `get` call from a version 2 caller to the version 2 object. The infrastructure ensures that such an object always exists by dynamically installing future-SOs and by only discarding past-SOs for dead versions.

Simulation objects may contain state and may use this state to implement calls. SOs must automatically recover their state after a node failure. Because they have state, SOs must run on the receiving side of method calls. The alternative (running the SO on the caller) would prevent callers from sharing the state of a single SO, since each caller would access the state of their own SO.

Another consequence of allowing SOs to have state is that their state must be initialized when the SOs are installed. Since past-SOs are installed when a node upgrades to a new version, they initialize their state from the old version's state, as depicted in Figure 4. Since future-SOs may be installed at any time, they initialize their state without any input. Both kinds of SOs may call methods on other nodes as part of initializing their state.

Notation SO_p^v denotes a past-SO that simulates a version v type by calling methods on a version $v+1$ type (the specific types are generally clear from context). SO_f^v denotes a future-SO that simulates a version v type by calling methods on a version $v-1$ type.

6.3 Examples

Since SOs are only required on the receiving end of method calls, we do not need to define any SOs for the clients in our example system. For each of the two example upgrades, we must define a past-SO and a future-SO for the servers, for a total of four SOs.

The first SO we consider is SO_f^2, the future-SO that lets version 1 servers support version 2 calls. Recall that version 2 adds the method `addComment(name, string)`, changes

[2]Past-SOs might never be discarded by systems in which upgrades happen very slowly (e.g., peer-to-peer).

the method `get(name)` to return a document and its comments, and leaves the method `set(name, string)` unchanged. SO_f^2 implements version 2's `set` by calling version 1's `set` method. SO_f^2 implements `addComment` by storing a mapping from names to comments in its own state (when it is installed, SO_f^2 initializes its state to the empty mapping). SO_f^2 implements `get` by calling version 1's `get` returning the concatenation of that result with the comments it has stored for that document. This example illustrates how SOs can use their state to implement more powerful behaviors than stateless handlers can.

The second SO we consider is SO_p^1, the past-SO that lets version 2 servers support version 1 calls. SO_p^1 implements version 1's `set` by calling version 2's `set`. SO_p^1 implements `get` by calling version 2's `get`, stripping out any comments, and returning the uncommented document. One might imagine avoiding having to strip out comments by storing documents in SO_p^1's state and getting them from there, but this would miss the effects of calls to `set` at version 2 and so would violate our requirement that the state of the node appear consistent at different versions.

The third SO we consider is SO_f^3, the future-SO that lets version 2 servers support version 3 calls. Recall that version 3 changes `set` so that it clears the comments for the document being set. One might hope to implement SO_f^3 by delegating all the version 3 methods to version 2 and, upon a `set`, calling some version 2 method to clear the comments for that document. Unfortunately, such a method may not be available. Even if it were, clearing the comments violates the expectations of version 2 clients (since version 2 servers never remove comments). But not clearing the comments violates the expectations of version 3 clients. One might try having SO_f^3 remember the comments at the time of the last `set` then strip out just those comments in `get`, but this would not work, since SO_f^3 would miss calls to `set` at version 2 and so might not remember the right comments. Even if this could work, it would cause an inconsistency between the state observed by version 2 and version 3 clients. Therefore, SO_f^3 may have to cause some calls to fail.

The fourth SO we consider is SO_p^2, the past-SO that lets version 3 servers support version 2 calls. SO_p^2 implements `set` and `addComment` by delegating to version 3. SO_p^2 cannot simply delegate `get` to version 3, since version 3 may clear the comments for a document, which would violate the expectations of version 2 clients. One might try having SO_p^2 remember all comments added for each document and return them in `get`, but this would miss calls to `addComment` at version 3 and so might miss some comments. Even if this could work, it would cause an inconsistency between the state observed by version 2 and version 3 clients. Therefore, SO_p^2 may have to cause some calls to fail.

These examples illustrate two points. First, SOs are limited by the fact that they can only see calls made at their own version. Second, some SO implementations can cause inconsistent views of a node's state at different versions. Section 8 elaborates on these points.

7 Transform Functions

Transform functions (TFs) are procedures defined by the upgrader to convert a node's persistent state from one version to the next. In previous systems [13,18,25], TFs converted the old object into a new one whose representation (a.k.a. "rep") reflected the state of the old one at the moment the TF ran. Our system extends this approach to allow the TF to also

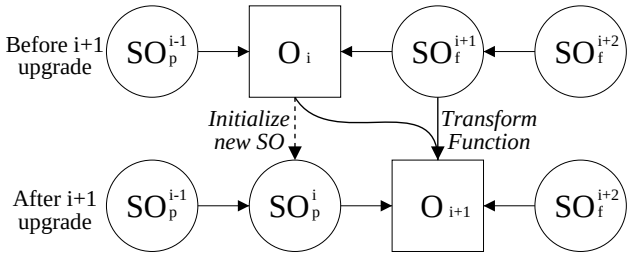


Figure 4: State transforms when upgrading to version i+1. Squares represent the states of instances; circles represent the states of simulation objects.

access the future-SO created for its version, as illustrated in Figure 4. The TF must then produce a new-class object whose state reflects both the state of the old object and the state of the future-SO. The upgrader can simplify the TF by making the future-SO stateless; then the TF's input is just the old version's state.

TFs may also call methods of the new version on other nodes. However, a TF cannot depend on these calls succeeding, since they may be simulated on the remote node and so may fail due to simulation limitations.

Previous systems provide tools to generate TFs automatically [23, 28, 41]. We believe such tools can be useful to generate simple TFs and SOs, but creating complex TFs and SOs will require human assistance.

Notation TF_v denotes a transform function that produces the state for a version v class from the state of a version $v-1$ class (the specific classes are generally clear from context).

7.1 Examples

Since we have not described any state for clients, we only consider the two TFs for the servers: TF_2, which produces version 2 server state from version 1, and TF_3, which produces version 3 server state from version 2.

TF_2 generates state for the version 2 server from the state of the version 1 server and the state of SO_f^2. We define these states as:

- **Version 1 state**: set of ⟨*name*, *document*⟩
- SO_f^2 **state**: set of ⟨*name*, list of *comment*⟩
- **Version 2 state**: set of ⟨*name*, *document*, list of *comment*⟩

TF_2 generates the version 2 state by joining the version 1 state with SO_f^2's state on *name*.

TF_3 generates state for the version 3 server from the state of the version 2 server and the state of SO_f^3. Since SO_f^3 has no state, TF_3 only considers the state of the version 2 server. The version 2 server state is the same as that of the version 3 server, so TF_3 just transforms the abstract value of this state from the version 2 representation to the version 3 representation [22]. For example, if version 2 stores its set using a list and version 3 uses a hash table, TF_3 populates the hash table with each element from the list.

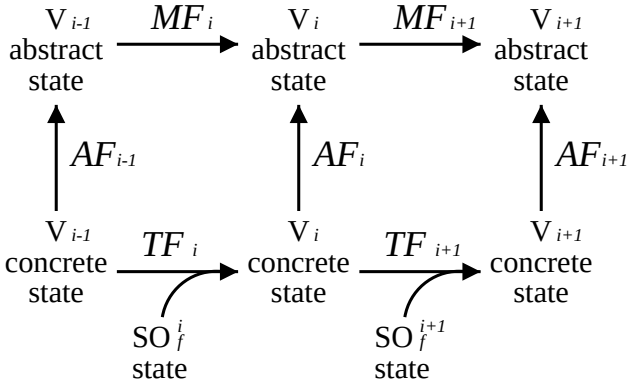


Figure 5: The relationship between transform functions, abstraction functions, and mapping functions.

8 Correctness Criteria

This section presents informal correctness criteria for simulation objects. The key question we are trying to answer is, when must an SO cause a call to fail? We describe the criteria in the context of a node running version i, O_i, with a single past-SO, SO_p^{i-1}, and a single future-SO, SO_f^{i+1}. We assume atomicity, i.e., calls to a node run in some serial order.[3]

There can be multiple clients of a node, and these clients may be running different versions. This means that calls to different versions can be interleaved. For example, a call to O_i (made by a client at version i) may be followed by a call to SO_f^{i+1} (made by a client at version $i+1$), which may be followed by a call to SO_p^{i-1} (made by a client running version $i-1$), and so on. We want this interleaving to make sense.

Also, clients running different versions may communicate about the state of a node. A client at version i may use (observe or modify) the state of the node via O_i, and a client at version $i+1$ may use the state of the node via SO_f^{i+1}, and the two clients may communicate about the state of the node out-of-band. We want the state of the node to appear consistent to the two clients.

We assume that each version has a specification that describes the behavior of its objects. In addition we require that version $i+1$'s specification explain how version $i+1$ is related to the previous version i. This explanation can be given in the form of a *mapping function*, MF_{i+1}, that maps the abstract state of O_i to that of O_{i+1}. An *abstraction function* maps an object's concrete state to its abstract state. Figure 5 depicts the relationship between transform functions, abstraction functions, and mapping functions.

We start with the obvious criteria: SO_f^{i+1}, O_i, and SO_p^{i-1} must each satisfy their version's specification. In the case of O_i we expect "full compliance," but in the case of the SOs, calls may fail when necessary, as defined in the following subsections.

[3]Our infrastructure can guarantee atomicity naïvely by locking the entire node each time it receives a call. This design is inefficient because it does not allow calls to execute concurrently even when this would be safe. Improving this design is future work.

8.1 Future SOs

This section discusses the correctness criteria for SO_f^{i+1}. We need to understand what each method of SO_f^{i+1} is allowed to do. The choices are: fail, access/modify the state of O_i, or access/modify the state of SO_f^{i+1}.

When going from version i to $i+1$, some state of O_i is reflected in O_{i+1}, and some is forgotten. We can view the abstract state of O_i as having two parts, a dependent part D_i and an independent part I_i, and the abstract state of O_{i+1} as having two parts, D_{i+1} and I_{i+1}. These parts are defined by MF_{i+1}: MF_{i+1} ignores I_i, uses D_i to produce D_{i+1}, and trivially initializes I_{i+1}.

Now we can describe the criteria for SO_f^{i+1}. A call to SO_f^{i+1} uses (observes or modifies) D_{i+1}, I_{i+1}, or both. Calls that use only I_{i+1} execute directly on SO_f^{i+1}'s rep. However, calls that use D_{i+1} must access O_i, or else clients at versions i and $i+1$ may see inconsistencies.

Consider our example upgrade from version 1 servers to version 2 servers. The abstract state of a version 1 server is a set of $\langle$*name*, *document*$\rangle$ pairs. The abstract state of a version 2 server is a set of $\langle$*name*, *document*, *comments*$\rangle$ tuples, where *comments* is a list of comments. So, the mapping function is:

$$MF_2 = \lambda set : \{\langle name, document\rangle\}.\{\langle n, d, [\,]\rangle | \langle n, d\rangle \in set\}$$

In this example, $D_1 = D_2 =$ the set of $\langle$*name*, *document*$\rangle$ pairs, and $I_2 =$ the set of $\langle$*name*, *comments*$\rangle$ pairs. Calls to SO_f^2 to add or view comments can be implemented by accessing SO_f^2's rep, where information about comments is stored, but calls that access the documents must be delegated to O_1. This way we ensure, e.g., that a modification of a document made via a call to SO_f^2 will be observed by later uses of O_1.

Thus we have the following condition:

1. Calls to SO_f^{i+1} that modify or observe D_{i+1} must be implemented by calling methods of O_i.

This condition ensures that modifications made via calls to SO_f^{i+1} are visible to users of O_i, and that modifications made via calls to O_i are visible to users of SO_f^{i+1}.

However, there is a problem here: sometimes it is not possible to implement calls on D_{i+1} by delegating to O_i. For example, consider a version 2 server, O_2, that supports version 3 using a future-SO, SO_f^3. MF_3 maps the abstract state of version 2 directly to version 3: $D_3 = D_2 =$ the set of $\langle$*name*, *document*, *comments*$\rangle$ tuples, so SO_f^3 must delegate all its methods to O_2 (note that this D_2 is different from the D_2 defined by MF_2). Version 3's specification requires that SO_f^3 clear the comments for a document when the document is `set`, but O_2 provides no methods to do this. SO_f^3 could make it appear as though the comments were removed by keeping track of removed comments in its own rep and "subtracting" those comments when calls observe D_3, but this creates an inconsistency between the states visible to version 2 and version 3 clients.

To prevent such inconsistencies, we require:

2. Calls to SO_f^{i+1} that use D_{i+1} but that cannot be implemented by calling methods of O_i must fail.

So calls to `set` on SO_f^3 must fail. SO_f^3 can still implement `get` and `addComment` by calling O_2.

These conditions disallow caching of mutable O_i state in SO_f^{i+1}. Caching of immutable state is allowed since no inconsistency is possible.

8.2 Past SOs

When O_i upgrades to O_{i+1}, a past-SO, SO_p^i, is created to handle calls to version i. We can apply all the above criteria to SO_p^i by substituting SO_p^i for SO_f^{i+1}, D_i for D_{i+1}, and O_{i+1} for O_i. In addition, SO_p^i must initialize its state from I_i, so that calls to SO_p^i that access I_i reflect the effects of previous calls to O_i.

Recall that $D_1 = D_2 =$ the set of $\langle name, document \rangle$ pairs, so SO_p^1 must implement `get` and `set` by delegating to O_2. Version 1's specification says `get` returns just the requested document, so SO_p^1 must strip the comments off any document returned by O_2. This does not create an inconsistency between the states observed by version 1 and version 2 clients, since version 1 clients no nothing about comments.

Similarly, SO_p^2 must delegate all its methods to O_3. Version 2's specification says comments are never removed, but version 3 allows O_3 to remove them. Therefore, SO_p^2 cannot implement `get` by delegating to O_3, so it must cause `get` to fail. If the node chains SO_p^1 to SO_p^2, then since SO_p^1 delegates `get` to SO_p^2, SO_p^1's `get` will also fail.

8.3 Transform Functions

TF_{i+1} produces a rep for O_{i+1} from that of O_i, i.e., it is the concrete analogue of MF_{i+1}. Where TF_{i+1} differs is in how it produces the concrete analogue of I_{i+1}: rather than initializing it trivially (as MF_{i+1} does), TF_{i+1} initializes I_{i+1} from the rep of SO_f^{i+1}. This ensures that calls to O_{i+1} that access I_{i+1} reflect the effects of previous calls to SO_f^{i+1}.

8.4 Limitations

Our correctness criteria only consider the relationship between *successive* versions and so cannot capture the relationship between two versions that are separated by one or more versions. For example, suppose versions 1 and 3 support `getValue()` and `setValue()`, but version 2 does not. This might happen if the upgrader mistakenly removes a feature then restores it in a later version. Ideally, modifications made via `setValue()` at version 1 would be visible via `getValue()` at version 3, and vice versa. But since SOs are only allowed to access the next successive version, they cannot implement these semantics.

We might fix this by extending our criteria to consider more than two versions at a time and by allowing SOs to access multiple previous versions. However, defining SOs in terms of multiple versions complicates reasoning about their correctness. This thesis will investigate whether these extensions can be practical.

Another limitation is that SOs only see calls made at their own version. This means SOs cannot implement behaviors that require action every time some method is called, e.g., logging calls for auditing or clearing comments for a document every time it is set. Removing this limitation, however, would complicate development of SOs, since developers would have

to reason about the effects of every method from every version on the state of the SO. This thesis will investigate whether there are simpler ways of working around this limitation.

9 Related Work

This section reviews related work in the context of our approach. For a more general treatment, see our annotated bibliography [9].

9.1 Upgrade Systems

Utilities like `rsync` [42] and package installers [1, 5, 7] automatically upgrade nodes over a network. Their upgrade scheduling takes two basic factors into account: the needs of the application (e.g., is an urgent security update pending?) and the needs of the user (e.g., is the user idle?). This suffices for many user applications, but does nothing to ensure that the system as a whole provides service.

Centrally-administered systems [4, 21, 39] do nothing special to handle upgrades that cause incompatibilities between versions. They enable centralized upgrade scheduling, which is useful for testing an upgrade on a few nodes or controlling the rate at which nodes upgrade, e.g., to manage bandwidth consumption.

Google [19] upgrades entire data centers at once and provides service by redirecting clients via DNS to other, redundant data centers (that are not upgrading). Since all the nodes in a data center upgrade together, nodes running different versions never communicate. Therefore, upgrades can change the protocols used within data centers but not those between data centers.

Reconfigurable distributed systems enable the replacement of object implementations in distributed object systems, such as Argus [13], Conic [27], Polylith [25], CORBA [11, 12], and Java [35]. They also let an administrator add and remove objects and change the links between them. These systems require manual upgrade scheduling and assume compatibility between versions.

9.2 State Transformation

Many upgrade systems allow the transformation of state between versions [13, 17, 18, 20, 23, 25, 41]. Most base the correctness of their transform functions on Herlihy and Liskov's value transmission technique for ADTs [22]. Some also provide tools that automatically generate transforms from the old and new version of an object's definition [23, 28, 41].

Bloom [13, 14] argues that value transmission is insufficient: an upgrade must preserve the history of operations on an object. For example, the current value of a random number generator captures the history of previous values yielded by the generator, so an upgrade must preserve this history and must not repeat any previous values.

9.3 Upgrade Scheduling

Existing work supports only simple kinds of upgrade scheduling, such as manual schedules, rate-controlled schedules [39], one-at-a-time upgrades of replicas [41], or opportunistic schedules that wait until a module quiesces [18]. Our work supports all these schedules and more by deploying upgrader-defined scheduling functions on nodes.

9.4 Mixed Mode Systems

Previous work has proposed using adapters to enable mixed mode systems to provide service, the most thorough discussion of which is Senivongse's [36]. Senivongse proposes several ideas that appear in our design, including the use of adapters to support new behavior on non-upgraded nodes. However, Senivongse does not discuss how these adapters correctly simulate one version's specification on another's, how they might be deployed, or how to manage their state.

PODUS [18] supports *interprocedures* that translate calls made to the old version of a procedure into calls to the new version. The Eternal system [41] supports *wrappers* that let nodes upgrade at different times by translating calls between them. However, neither of these projects uses simulation to support future behavior or discusses how wrappers can correctly implement one version's specification using another's.

Object-oriented databases that use *schema versioning* may contain contain instances of different versions of a type. Furthermore, they provide ways for an instance to handle accesses from programs using a different (older or newer) version of its type. The ENCORE system [40] wraps each instance with a version set interface (VSI) that can accept any method call for any version of the instance's type. If the instance cannot satisfy a call directly, handlers in the VSI can substitute a response. Unfortunately, the number of handlers grows quadratically in the number of versions: each time a new version is defined, new handlers must be defined for each old version. Monk and Sommerville's system [32] uses "update" and "backdate" functions to convert calls from any version to the version of the instance. These functions can chain together, so new version requires only two new functions.

Federated distributed systems [16,31,34,37] address similar challenges as those posed by mixed mode systems. Federated systems must transform calls or data as they pass from one domain of the system to another, e.g., a system may transform a monetary value from one currency to another as it is transferred between different countries. Elements of our techniques for handling mixed mode systems may also apply to federated systems.

9.5 Dynamic Updating

Dynamic updating [17,18,20,23,24,30] enables nodes to upgrade with minimal service interruption. This is complementary to our approach and can reduce downtime during upgrades.

10 Thesis Schedule

In her keynote address at SOSP 2001, Prof. Liskov outlined our techniques for supporting upgrades in distributed systems [29]. In Summer 2002, we reviewed the literature on software

upgrade systems [9]. In Fall 2002, we did paper studies of several scenarios using our model [8]. In January 2003, we refined our design and submitted a position paper to the HotOS 2003 workshop, which has since been accepted [10]. The roadmap for the rest of the thesis is:

Spring 2003 Formalize the correctness criteria for simulation objects, transform functions, and scheduling functions. (This thesis will not attempt to generate any of these objects or check their correctness automatically.) Prototype the infrastructure using Sun RPC.

Summer 2003 Complete the prototype and evaluate basic upgrade scenarios ("document server" and "new DHash block type"). Deploy on PlanetLab [33] for wide-area evaluation.[4] Revise the design as needed.

Fall 2003 Finalize the design and implementation. Evaluate upgrade scenarios (the two above plus "NFS3 to NFS4" and a simulated sensor net example). If time permits, add support for message-passing and other RPC or RMI protocols.

Winter and Spring 2004 Write thesis. If time permits, submit a paper on the architecture to OSDI.

11 Required Facilities

For development and local evaluation of our prototype, the Programming Methodology Group machines and software will suffice. For wide-area evaluation, we plan to use PlanetLab, which is currently free for academic use.

12 Future Work

We believe that the design sketched above is a good starting point for supporting automatic upgrades for distributed systems, but much work remains to be done. Here are some of the more interesting open problems that we hope to address:

- Formalize correctness criteria for scheduling functions, simulation objects, and transform functions.
- Provide support for nodes that communicate by message passing rather than by RPC or RMI.
- Provide support for multiple objects per node.
- Investigate ways to run transform functions lazily, so that a node can upgrade to the next version quickly and add additional information to its representation as needed.
- Investigate ways to recover from upgrades that introduce bugs. One possibility is to use a later upgrade to fix an earlier, broken one. This requires a way to undo an upgrade, fix it, then somehow "replay" the undone operations [15].

[4] PlanetLab developer Brent Chun is interested in using our methodology to upgrade PlanetLab services and possibly even PlanetLab's own infrastructure. A cooperative effort may make sense here.

- Investigate ways to allow the upgrade infrastructure itself to be upgraded.

We are currently implementing a prototype of our upgrade infrastructure for RPC-based systems [?]. We plan to use the prototype to evaluate upgrades for several systems, including Chord and NFS.

References

[1] APT HOWTO. `http://www.debian.org/doc/manuals/apt-howto/`.

[2] Cisco Resource Manager. `http://www.cisco.com/warp/public/cc/pd/wr2k/rsmn/`.

[3] EMC OnCourse. `http://www.emc.com/products/software/oncourse.jsp`.

[4] Marímba. `http://www.marimba.com/`.

[5] Red Hat up2date. `http://www.redhat.com/docs/manuals/RHNetwork/ref-guide/up2date.html`.

[6] Windows 2000 clustering: Performing a rolling upgrade. 2000.

[7] Managing automatic updating and download technologies in Windows XP. `http://www.microsoft.com/WindowsXP/pro/techinfo/administration/manageau%toupdate/default.asp`, 2002.

[8] S. Ajmani. Distributed system upgrade scenarios. Oct. 2002.

[9] S. Ajmani. A review of software upgrade techniques for distributed systems. Aug. 2002.

[10] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: How to upgrade distributed systems. In *HotOS-IX*, May 2003.

[11] J. P. A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for CORBA, 2001.

[12] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *4th Intl. Conf. on Configurable Dist. Systems*, pages 35–42, Annapolis, MD, May 1998.

[13] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System.* PhD thesis, MIT, 1983. Also available as MIT LCS Tech. Report 303.

[14] T. Bloom and M. Day. Reconfiguration in Argus. In *Intl. Workshop on Configurable Dist. Systems*, pages 176–187, London, England, Mar. 1992. Also in [26], pages 102–108.

[15] A. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to dependability. In *10th ACM SIGOPS European Workshop*, Saint-Emilion, France, Sept. 2002.

[16] H. Evans and P. Dickman. DRASTIC: A run-time architecture for evolving, distributed, persistent systems. *Lecture Notes in Computer Science*, 1241:243–??, 1997.

[17] R. S. Fabry. How to design systems in which modules can be changed on the fly. In *Intl. Conf. on Software Engineering*, 1976.

[18] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, pages 111–128, Feb. 1991.

[19] S. Ghemawat. Google, Inc., personal communication, 2002.

[20] D. Gupta and P. Jalote. On-line software version change using state transfer between processes. *Software Practice and Experience*, 23(9):949–964, Sept. 1993.

[21] R. S. Hall, D. Heimbeigner, A. van der Hoek, and A. L. Wolf. An architecture for post-development configuration management in a wide-area network. In *Intl. Conf. on Dist. Computing Systems*, May 1997.

[22] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, 1982.

[23] M. W. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–23, 2001.

[24] G. Hjalmtysson and R. Gray. Dynamic C++ classes—A lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conf.*, pages 65–76, June 1998.

[25] C. R. Hofmeister and J. M. Purtilo. A framework for dynamic reconfiguration of distributed programs. Technical Report CS-TR-3119, University of Maryland, College Park, 1993.

[26] *IEE Software Engineering Journal, Special Issue on Configurable Dist. Systems.* Number 2 in 8. IEE, Mar. 1993.

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

[28] B. S. Lerner. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1):83–127, 2000.

[29] B. Liskov. Software upgrades in distributed systems, Oct. 2001. Keynote address at the 18th ACM Symposium on Operating Systems Principles.

[30] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *European Conf. on Object-Oriented Programming*, 2000.

[31] B. Meyer, S. Zlatintsis, and C. Popien. Enabling interworking between heterogeneous distributed platforms. In *IFIP/IEEE Intl. Conf. on Dist. Platforms (ICDP)*, pages 329–341. Chapman & Hall, 1996.

[32] S. Monk and I. Sommerville. A model for versioning of classes in object-oriented databases. In *Proceedings of BNCOD 10*, pages 42–58, Aberdeen, 1992. Springer Verlag.

[33] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *In Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, Oct. 2002. PlanetLab.

[34] P. Reichl, D. Thißen, and C. Linnhoff-Popien. How to enhance service selection in distributed systems. In *Intl. Conf. Dist. Computer Communication Networks—Theory and Applications*, pages 114–123, Tel-Aviv, Nov. 1996.

[35] T. Ritzau and J. Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, May 2000.

[36] T. Senivongse. Enabling flexible cross-version interoperability for distributed services. In *Intl. Symposium on Dist. Objects and Applications*, Edinburgh, UK, 1999.

[37] T. Senivongse and I. Utting. A model for evolution of services in distributed systems. In S. Schill, Mittasch and Popien, editors, *Distributed Platforms*, pages 373–385. Chapman and Hall, Jan. 1996.

[38] L. Sha, R. Rajkuman, and M. Gagliardi. Evolving dependable real-time systems. Technical Report CMS/SEI-95-TR-005, CMU, 1995.

[39] M. E. Shaddock, M. C. Mitchell, and H. E. Harrison. How to upgrade 1500 workstations on Saturday, and still have time to mow the yard on Sunday. In *Proc. of the 9th USENIX Sys. Admin. Conf.*, pages 59–66, Berkeley, Sept. 1995. Usenix Association.

[40] A. H. Skarra and S. B. Zdonik. The management of changing types in an object-oriented database. In *OOPSLA*, pages 483–495, 1986.

[41] L. A. Tewksbury, L. E. Moser, and P. M. Melliar-Smith. Live upgrades of CORBA applications using object replication. In *IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 488–497, Florence, Italy, Nov. 2001.

[42] A. Trigdell and P. Mackerras. The rsync algorithm. Technical report, 1998. `http://rsync.samba.org`.