USENIX Association

Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems

Lihue, Hawaii, USA
May 18–21, 2003

Scheduling and Simulation: How to Upgrade Distributed Systems

Sameer Ajmani Barbara Liskov
Massachusetts Institute of Technology
Laboratory for Computer Science
{ajmani,liskov}@lcs.mit.edu

Liuba Shrira
Brandeis University
Computer Science Department
liuba@cs.brandeis.edu

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

Long-lived distributed systems, like server clusters, content distribution networks, peer-to-peer systems, and sensor networks, require changes (upgrades) in their software over time to fix bugs, add features, and improve performance. These systems are large, so it is impractical for an administrator to upgrade nodes manually (e.g., via remote login). Instead, upgrades must propagate automatically, but the administrator may still require control over the order and rate at which nodes upgrade to avoid interrupting service or to test an upgrade on a few nodes. Thus, upgrades may happen slowly, and there may be long periods of time when some nodes are upgraded and others are not. Nonetheless, the system as a whole should continue to provide service.

The goal of our research is to support automatic upgrades for such systems and to enable them to provide service during upgrades. Earlier approaches to automatically upgrading distributed systems [9–11, 17, 18, 22] or distributing software over networks [1–6, 8, 15, 25] do little to ensure continuous service during upgrades. The Eternal system [27], the Simplex architecture [24], and Google [14] enable specific kinds of systems to provide service during upgrades, but they do not provide general solutions.

An automatic upgrade system must:

- propagate upgrades to nodes automatically
- provide a way to control *when* nodes upgrade
- enable the system to provide service when nodes are running different versions
- provide a way to preserve the persistent state of nodes from one version to the next

To address these requirements, our approach includes 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.

Scheduling functions (SFs) are procedures that run on nodes and tell them when to upgrade. SFs can implement a variety of upgrade scheduling policies.

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 [13, 23, 27], ours includes correctness criteria to ensure that simulation objects reflect node state consistently across different versions. These criteria require that some interactions 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. Our contribution is to show how TFs interact with SOs to ensure that nodes upgrade to the correct new state.

Our approach takes advantage of the fact that long-lived systems are *robust*. They tolerate node failures: nodes are prepared for failures and know how to recover to a consistent state. This means that we can model a node upgrade as a soft restart. Robust systems also tolerate communication problems: remote procedure calls may fail, and callers know how to compensate for such failures. This means that we can use a failure response when calls occur at inopportune times, e.g., when a node is upgrading or when a node's simulation object is unable to carry out the requested action.

The rest of the paper is organized as follows. Section 2 presents our upgrade model and assumptions. Section 3 describes the upgrade infrastructure; Section 4, scheduling functions; Section 5, simulation objects; and Section 6, transform functions. Section 7 presents correctness criteria, and Section 8 concludes.

2 Model and Assumptions

We model each node as an object that has an identity and a state; we assume there is just one object per node but the model can be extended to handle multiple objects. Objects are fully-encapsulated; inter-object interaction is by means of method calls. Systems based on remote procedure calls [26] or remote method invocations [21] map easily to this model; extending the model to message-passing [19] is 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.

Each node runs a top-level class—the class that implements its object. We assume class definitions are stored in well-known repositories and define the full implementation of a node, including its subcomponents and libraries. Different nodes are likely to run different classes, e.g., clients run one class, while servers run another.

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. 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* 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 two versions: the new one and the preceding one.

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., 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 1: 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 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.

When an upgrade layer hears about a new version, it notifies the node's *upgrade manager (UM)*. The UM 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 UM fetches the appropriate class upgrade and class implementation from the upgrade server. The UM 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 UM 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 UM 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 UM shuts down the current node software, causing it to persist

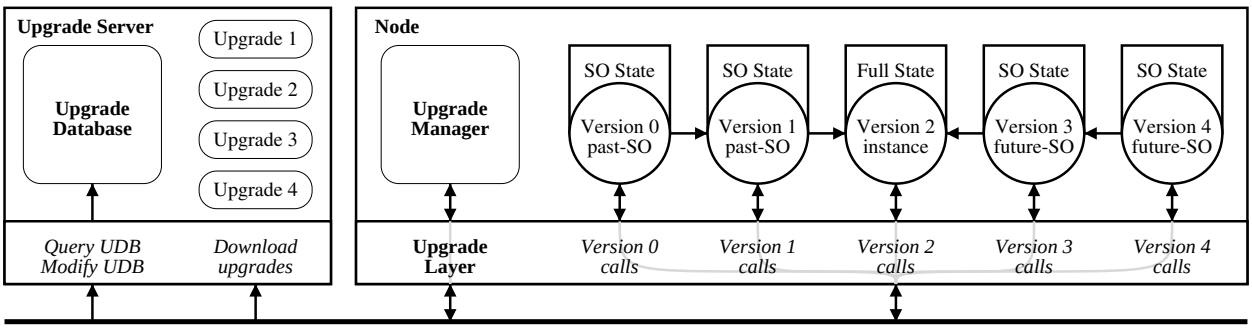

Figure 1: *An example system that consists of an upgrade server and a single system node. Arrows indicate the direction of (remote) method calls. The upgrade server stores the upgrade database and publishes upgrades for versions 1 through 4. The node is running version 2, has a chain of past-SOs to support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

its state. The UM 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 UM then discards the future-SO and installs the past-SO, which lets the node continue to support the previous version. Finally, the UM starts the new version's software, which recovers from the newly-transformed persistent state.

4 Scheduling Functions

Scheduling functions (SFs) are procedures defined by the upgrader that tell nodes when to upgrade. Unlike existing systems that coordinate upgrades centrally [4,15,25], 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 [7, 27].

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 [25].

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 the 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.

The main challenge in designing scheduling functions is ensuring that they behave correctly. Since SFs control the rate at which node upgrades occur, they can affect a system's availability, fault-tolerance, and performance. We are investigating ways to reason about SF correctness and their system-wide effects.

5 Simulation Objects

Simulation objects (SOs) are defined by the upgrader to enable communication between nodes running differ-

ent versions. This is necessary when nodes upgrade at different times, since nodes running older versions may make calls on nodes running newer versions, and vice versa. It is important to enable simulation in both directions, because otherwise a slow upgrade can partition upgraded nodes from non-upgraded ones (since calls between those nodes will fail). Simulation also simplifies software development by allowing implementors to write their software as if every node in the system were running the same version.

SOs are wrappers: they delegate (most of) their behavior to other objects. This means that SOs are simpler to implement than full class implementations, but they are also slower than full implementations and may not be able to implement full functionality (as discussed in Section 7). If a new version does not admit good simulation, the upgrader may choose to use an eager upgrade schedule (as discussed in Section 4) and avoid the use of SOs altogether—but the upgrader must bear in mind that an eager schedule can disrupt service.

An upgrader defines two simulation objects for each version, a *past-SO* and a *future-SO*. A *past-SO* implements an old version 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.

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 1. An SO may call methods on the next object in the chain; it is unaware of whether the next object is the current object or another SO. When a node receives a call, its upgrade layer dispatches the call to the object that implements the version of that call. 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. When an SO is installed, it must initialize its state. Past-SOs initialize their state from the old version's persistent state, as depicted in Figure 2. Future-SOs initialize their state without any input.

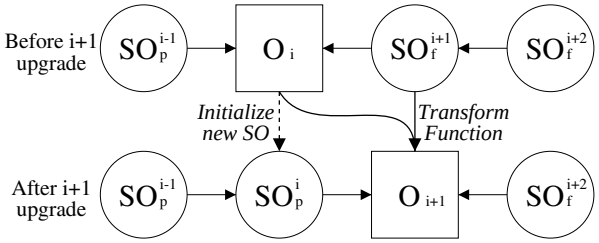

Figure 2: *State transforms for upgrading from version i to version i+1. Squares represent the states of the current version; circles represent the states of simulation objects.*

6 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 [11, 13, 17], 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 access the future-SO created for its version, as illustrated in Figure 2. 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.

In systems that enable nodes to recover their persistent state from other nodes, the TF may be able to simply discard a node's state and rely on state recovery to restore it. This requires that state transfer work correctly between nodes running different versions (e.g., using SOs) and that the scheduling function allow enough time between node upgrades for state transfer.

Previous systems provide tools to generate TFs automatically [16, 20, 27]. We believe such tools can be useful to generate simple TFs and SOs, but creating complex TFs and SOs will require human assistance.

7 Correctness Criteria

This section presents informal correctness criteria for simulation objects. 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.

We assume that each version $i+1$ has a specification that describes the behavior of its objects. In addition we require that the 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}.

We have 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. One of the main questions we are trying to answer is, when is failure necessary?

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 $i\quad 1$), 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.

7.1 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 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.

For example, suppose O_i and O_{i+1} are web servers, and O_{i+1} adds support for comments on web pages. MF_{i+1} produces O_{i+1}'s pages from O_i's pages: D_{i+1} = D_i = the pages. O_{i+1}'s comments for each page are independent of O_i's state, i.e., I_{i+1} = the comments. Calls to SO_f^{i+1} to add or view comments can be implemented by accessing SO_f^{i+1}'s rep, where information about comments is stored, but calls that access the pages must be delegated to O_i. This way we ensure, e.g., that a modification of a page made via a call to SO_f^{i+1} will be observed by later uses of O_i.

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. Suppose O_i is an Archive (a set that can only grow) and O_{i+1} is a Cache (a set that can grow and shrink). Then, D_{i+1} = D_i = the set. SO_f^{i+1} cannot implement removals by delegating to O_i, because O_i does not support removals. SO_f^{i+1} could support removals by keeping track of removed elements in its own rep and "subtracting" these elements when calls observe D_{i+1}, but this creates an inconsistency between the states visible to clients at version i and $i+1$. 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 on SO_f^{i+1} for removals must fail. SO_f^{i+1} can still implement additions and fetches by calling O_i.

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.

7.2 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.

7.3 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 Future Work

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

- 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 [12].
- 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 [26]. We plan to use the prototype to evaluate upgrades for several systems, including Chord and NFS.

Acknowledgments

This research is supported by NSF Grant IIS-9802066 and by NTT. The authors thank George Candea, Alan Donovan, Michael Ernst, Anjali Gupta, Chuang-Hue Moh, Steven Richman, Rodrigo Rodrigues, Emil Sit, and the anonymous reviewers for their helpful comments.

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] Rsync. `http://www.rsync.org/`.

[7] Windows 2000 clustering: Performing a rolling upgrade. `http://www.microsoft.com/windows2000/techinfo/planning/incremental/roll%upgr.asp`, 2000.

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

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

[10] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *4th Intl. Conf. on Configurable Dist. Systems*, 1998.

[11] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, 1983. Also as MIT Laboratory for Computer Science Technical Report 303.

[12] A. Brown and D. A. Patterson. Rewind, repair, replay: Three R's to dependability. In *10th ACM SIGOPS European Workshop*, 2002.

[13] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, 1991.

[14] S. Ghemawat. Google, inc., personal comm., 2002.

[15] R. S. Hall et al. An architecture for post-development configuration management in a wide-area network. In *Intl. Conf. on Dist. Computing Systems*, 1997.

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

[17] 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.

[18] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11), 1990.

[19] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Proc. 2nd Intl. Symposium on Operating Systems*, volume 13, 1979.

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

[21] Sun Microsystems. Java RMI specification, 1998.

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

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

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

[25] 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.*, 1995.

[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 *IEEE Intl. Conf. on Software Maintenance*, 2001.