A Review of Software Upgrade Techniques for Distributed Systems

Sameer Ajmani
ajmani@csail.mit.edu

MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street G908, Cambridge, MA 02139, USA
First Draft: August 7, 2002
Last Modified: November 7, 2004

Abstract

This document reviews work on software upgrades in distributed systems. The main text describes the work in various sub-areas, and the bibliography provides annotations on these and several other related papers.

1 Reconfigurable Distributed Systems

The earliest work on upgrades in distributed systems appears to be Bloom's work on reconfiguration in Argus [25,26]. Argus is a strongly-typed distributed system in which modules called "guardians" implement interfaces composed of sets of "handlers" (i.e., RPCs). Argus guarantees that stable state survives crashes and that actions are atomic. Bloom's work addresses the problem of replacing implementations in this environment. The unit of replacement may be a single guardian (which resides at a single physical node) or a set of multiple guardians, called a subsystem (which may span nodes). Bloom defines a formal model to determine which replacements are legal; these are those replacements that preserve or invisibly extend the replaced subsystem's continuation abstraction. Bloom presents several examples of replacements that seem intuitively legal but that actually violate this condition. The actual mechanisms used to replace subsystems allow a user to manually locate, add, remove, and restart guardians; get, put, and optionally translate guardian state; and fetch and rebind handlers. A sequence of these actions compose a replacement transaction; this transaction either may wait until it can acquire exclusive lock on all required guardians or may preempt and abort other clients' transactions. Bloom does not detail an implementation and cites the need for a higher-level user interface to replacement.

Kramer and Magee [62–64] describe how to upgrade a distributed system that is specified in Conic, i.e., as a set of modules and connections between them. Upgrades are specified as declarative change commands (link, unlink, create, and delete); a Configuration Manager (CM) translates these commands into a "change transaction" and executes it (but does not guarantee atomicity). The authors' earlier work [62] makes no assumptions about applications and so cannot guarantee application consistency after reconfiguration. Their later work [63] assumes applications can become "passive" and describes how to preserve consistency across reconfiguration by passivating the appropriate modules. The authors mention that recovery could be used instead of quiescence, but

they argue that this complicates applications. The system requires that the initiator of a transaction be informed of when a transaction completes (i.e., no one-way messages). The system does not support state transfer between old and new components, but the authors mention that this is possible between quiescent components.

Frieder and Segal [40,41,81,82] describe how to upgrade a procedure-oriented system (including RPC-based distributed systems) by replacing individual procedure definitions. A procedure P may be replaced only if P is inactive, i.e., (1) P is not on the runtime stack, (2) P's new version does not call any active procedure, and (3) no procedure semantically-dependent on P is active. Properties (1) and (2) are inferred from program syntax; property (3) must be supplied by the programmer. If P contains static state, the upgrader must provide a "mapper procedure" to convert the old version's state to the new version. If the upgrade changes P's interface, the upgrader must provide an "interprocedure" to convert old calls to new ones. The authors argue that this scheme works well for programs written in a top-down manner, since lower-level procedures become inactive often and are exactly those most likely to change (since they implement low-level program details). Thus, this scheme has difficulty upgrading long-running procedures and cannot upgrade procedures that run forever.

Hofmeister and Purtilo [52–54] extend the work of Kramer and Magee by implementing reconfiguration in Polylith, a module interconnection language. This work adds support for capturing and restoring process state via an intermediate abstract representation [49]. Reconfiguring modules must be quiescent; messages received during the reconfiguration are buffered and are used to initialize the replacement module. The authors also describe a packager, Surgeon, that determines how components should be integrated, generates interface software to connect components, and creates configuration commands to build the application. A "catalyst" module on each node actually runs reconfigurations using a package. Packaging may require component participation to save the state, transform state, restart the component, or delay upgrades until a suitable point (so that consistency can be maintained).

Barbacci et al. [19] describe reconfiguration in Durra, a module interconnection language. Durra reconfiguration is "triggered" by certain events, such as component failures. When such an event is detected by the local "cluster manager", the manager executes the appropriate reconfiguration.

Sha et al. [85] describe the Simplex architecture for supporting evolution of real-time systems that use commercial off-the-shelf (COTS) components. Upgrades are supported by grouping a set of analytically redundant components (i.e., that satisfy the same abstract spec) into a subsystem module. Each module contains a safety component, a baseline component, and an optional new component. A module manager monitors the behavior of the new component and, if it behaves correctly, replaces the baseline component with the new one.

Bidan et al. [24] describe a "Dynamic Reconfiguration Manager" for Aster, a CORBA-based distributed system. Rather than passivating objects as in previous work, the DRM passivates the links between objects. The authors define formal consistency and efficiency constraints for reconfiguration, and argue that their reconfiguration algorithm is optimally efficient (i.e., causes minimal disruption). Objects must implement a "Reconfigurable" interface to support dynamic reconfiguration.

Ritzau and Andersson's JDrums system [80] uses lazy upgrades to convert Java classes and objects to new versions. An upgrade consists of a class converter that converts static class data and an object converter that converts instances. The system uses a modified JVM to keep old versions of classes and objects around so that old references continue to work. The authors do

not comment on the problem of state divergence between different versions of the same object. They use Jini to deploy upgrades in distributed systems but have no mechanism to synchronize or otherwise schedule distributed upgrades.

Almeida et al. [15, 16] describe dynamic reconfiguration in CORBA. The main differences between their work and that of Bidan et al. are (1) support for re-entrant invocations, (2) support for atomic replacement of multiple objects, and (3) greater transparency using ORB extensions.

Tewksbury et al. [90] describe upgrades of CORBA applications in the Eternal system. They use replication of server objects to provide uninterrupted service during upgrades. Objects are replaced one-by-one with an intermediate version that implements both the old and the new behavior. Rather than allowing the system to exist in a hybrid state (i.e., where different objects are at different versions), the upgrade executes an "atomic switchover" that changes all objects from one version to another. Reliable, totally-ordered multicast ensures atomicity. If the upgrade changes an objects' interface, all clients that use that interface must are also be upgraded atomically. Furthermore, all affected objects must be quiescent when the switchover occurs (it's not clear how they manage "uninterrupted service" here!). The system uses wrapper functions to loosen this quiescence requirement (i.e., by translating old calls to new ones). The system provides an "upgrade preparer" tool that automatically generates wrapper functions and state transformers given the old and new versions of an object's code.

Solarski and Meling [88] propose upgrading a replicated service by upgrading each replica in turn, thus maintaining online service. They assume that (1) multiple upgrades do not interleave, (2) version v+1 offers a compatible interface to version v, (3) there exists a mapping from version v's state to version v+1, and (4) clients only use extensions offered by v+1's interface after all replicas are upgraded. Upgrades are scheduled by totally ordering the replicas (using some replica identifier). They assume full compatibility between different versions, so they don't need to convert messages between versions (i.e., no simulation mode). As this is just a position paper, it does not provide any design or implementation details.

2 Extensible Distributed Systems

Oki et al. [72] describe the "Information Bus", a publish-subscribe architecture to support extensible distributed systems. Components communicate by subscribing to data published on the bus; data is identified using application-specific hierarchical names. Thus, new services can replace old ones simply by taking over data publication. Data objects are dynamically typed, so new object implementations can be deployed as subtypes of existing types. The system supports legacy applications and services using "adapters" to translate between the application domain and the Information Bus object format.

Govindan et al. [44] describe active distributed services (ADS), which are systems composed of cooperating agents located on nodes on a network. Agents are extensible by plugging in new event handlers (e.g., message handlers). Handlers are pushed from agent to agent or are retrieved on demand (e.g., when a new event is encountered). Each physical node has one "actuator" that sends and receives handlers (cf. upgrade dissemination). Each ADS agent on a node has an "envoy" that uses the actuator to fetch new handlers. Together, the actuator and the envoys compose an ADS node's runtime "substrate."

3 Automatic Software Deployment

The Software Dock [46, 47] is an architecture for automatic software dissemination and updating. Software producers run servers called "release docks" that maintain a registry of their software releases and notify subscribers of new updates. Software consumers run "field docks" that maintain a registry of locally-installed software. Users download and authorize installation agents that in turn install the requested software and install update agents. A global event system routes update notifications to update agents, that in turn download and install updates (automatically resolving any dependencies). Local agents enforce access control to local resources, like the file system.

Weiler [94] argues that fully automatic upgrades are dangerous, since automatic upgrade mechanisms often introduce incompatibilities. Part of the problem is that each vendor develops and deploys their own automatic upgrade system; so standards could help fix this. Another problem is that it's hard for users to determine the cause of a problem; a log of installations and updates would help, and upgrade rollback may be necessary to fix such problems.

4 Security and Interoperability

Devanbu et al. [35] identify security issues for automatic software management. The system must protect the *integrity* of the software being shipped from vendor to user, the user's configuration, and messages from user to vendor that describe configurations. The system must *authenticate* software vendors and licensed software users. The system must protect the *privacy* of software components (because of their intellectual property value) and of software configurations (because they may reveal sensitive data). Finally, users must be able to *delegate* authorization, e.g., so that users can delegate configuration control to vendors and so that vendors can delegate configuration checking to a testing lab.

Senivongse [83, 84] describes how to use mappers to enable cross-version interoperation during distributed upgrades. By allowing nodes of different versions to communicate, one can change interfaces during an upgrade without interrupting service (unlike systems that atomically change all clients and servers). The author addresses many issues regarding the use of mappers: their applicability and limitations, automatic generation of mappers, mapper chains, backward mappers (new-to-old), optimizations, and issues that occur due to object subtyping.

References

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

[2] Battle.net multiplayer online game server. `www.battle.net`.

> A game server system that requires clients to version sync with the server before allowing them to set up games with other clients. This ensures that clients in the same game are version-synced, but also enables Battle.net to upgrade without interrupt existing games.

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

> A commercial, web-based management system that supports software upgrades for "Cisco switches, access servers, and routers."

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

A commercial "application that enables secure, reliable, automated distribution of files between systems across IP networks."

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

A peer-to-peer file sharing system that deploys upgrades eagerly: nodes version sync on eahc communication, so if one node upgrades, it causes each node it talks to to upgrade (epidemic dissemination).

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

A commercial system for "managing software on desktops, laptops, servers, and devices."

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

[8] Star wars galaxies multiplayer online game. `http://starwarsgalaxies.station.sony.com/`.

A multiplayer online game that requires clients to version sync with the server before logging in. Presumably, the server kicks off all clients (requiring them to resync) when a new version is deployed.

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

Describes how to maintain service while upgrading a cluster of NT or Windows 2000 servers with a new service pack. Servers are upgraded one-at-a-time, and resources and clients automatically fail over and fail back between nodes in the cluster, even when those nodes are running different versions (i.e., the cluster is in mixed mode).

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

[11] Sameer Ajmani. Distributed system upgrade scenarios. October 2002.

[12] Sameer Ajmani. A review of software upgrade techniques for distributed systems. August 2002.

[13] Sameer Ajmani. *Automatic Software Upgrades for Distributed Systems*. PhD thesis, MIT, September 2004.

[14] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Scheduling and simulation: How to upgrade distributed systems. In *HotOS-IX*, May 2003.

[15] Joao Paulo Almeida. An approach to dynamic reconfiguration of distributed systems based on object-middleware, May 2001.

See "Transparent Dynamic Reconfiguration" (Almeida et al.) for details. This paper provides more comparisons with related work.

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

Uses ORB extensions to intercept requests and thus passivate objects for reconfiguration. Request interceptors queue requests made during a reconfiguration. A "reconfiguration manager" handles the creation and deletion of objects, state transfer (and translation), and object passivation. A "location agent" provides indirection between clients and server objects; this allows clients to locate objects that migrate during a reconfiguration. The main differences between this and Bidan et al. are (1) support for re-entrant invocations, (2) support for atomic replacement of multiple objects, and (3) greater transparency using ORB extensions.

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

Describes how the O2 ODBS supports object-oriented views and data updates made to those views. A view is a virtual schema and base (set of persistent roots), defined using a view definition language (VDL). The view is expressed in terms of the root (underlying) schema and base. The paper defines completeness and consistency constraints on the view, and requires that updates made to the view respect those constraints. An update to a data attribute in a view is translated to updates on root data as follows: if an attribute in the root is the same (or renamed) in the view, then updates to the attribute view are made to the root. If the attribute in the view is defined as a function of the root attribute, and if the function is invertible, then an update is applied to the root by first applying the inverse of that function (e.g., if the view is root x 2, then new-root is new-view / 2). If the function is not invertible, then the user may define code that translates view updates to root updates. A runtime check made after such updates ensures that the new root value still maps to the new view value (if this check fails, the update aborts). Finally, if no translation is possible, the update fails.

[18] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma M. Da Silva, Orran Krieger, David J. Edelsohn Marc A. Auslander, Ben Gamsa, Gregory R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic behavior in systems software with hot-swapping. *IBM Systems Journal*, 42(1), 2003.

[19] M. Barbacci et al. Building fault-tolerant distributed applications with Durra. In Intl. Conf. on Configurable Dist. Systems [57], pages 128–139. Also in [55], pages 83–94.

Durra describes an application as a set of components (application tasks and communication channels), a set of alternative configurations showing how these components are connected at runtime, and a set of conditional configuration transitions that take place at runtime. A "cluster" is a physical grouping of components at a node; a "cluster manager" is responsible for starting and terminating application processes and links, for passing messages between components, for monitoring reconfiguration conditions, and for carrying out reconfigurations. The Durra runtime requires that processes be quiescent before reconfiguring; Durra relies on processes to declare themselves quiescent explicitly by making a call to their cluster managers. If a process does not quiesce in a timely manner, the cluster manager times out. While Durra tolerates component failures, it does not tolerate cluster failures.

[20] Donnie Barnes. RPM HOWTO. `http://www.rpm.org/RPM-HOWTO/`, November 1999.

[21] T. Bartoletti, L. A. Dobbs, and M. Kelley. Secure software distribution system. In *Proc. 20th NIST-NCSC National Information Systems Security Conf.*, pages 191–201, 1997.

Authenticating and upgrading system software plays a critical role in information security, yet practical tools for assessing and installing software are lacking in today's marketplace. the Secure Software Distribution System (SSDS) will provide automated analysis, notification, distribution, and installation of security patches and related software to network-based computer systems in a vendor-independent fashion. SSDS will assist with the authentication of software by comparing the system's objects with the patch's objects. SSDS will monitor vendors' patch sites to determine when new patches are released and will upgrade system software on target systems automatically. This paper describes the design of SSDS. Motivations behind the project, the advantages of SSDS over existing tools as well as the current status of the project are also discussed.

[22] Luc Bellissard, Slim Ben Atallah, Fabienne Boyer, and Michel Riveill. Distributed application configuration. In *Intl. Conf. on Dist. Computing Systems*, pages 579–585, 1996.

Olan is a module interconnection language (MIL) that extends the work of Conic, Polylith, Darwin, and Durra. In particular, Olan adds support for encapsulating legacy applications

as components of the system. Uses "interactions" to define functional dependencies between components and "connectors" to decouple implementations from interactions.

[23] Robert P. Bialek. The architecture of a dynamically updatable, component-based system. In Workshop on Dependable On-line Upgrading of Dist. Systems [97].

Combines dynamic component architectures (e.g., CORBA, EJB, DCOM) with dynamic software updating (e.g., Gupta and Jalote, Hicks) to create an architecture that supports both structural reconfigurations and non-stop updates to component implementations. Defines an "update descriptor" as a set of requests to add, remove, and/or update objects in the system, along with the replacement implementations (much like a package).

[24] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In Intl. Conf. on Configurable Dist. Systems [60], pages 35–42.

Defines a Dynamic Reconfiguration Manager (DRM) that coordinates reconfigurations in Aster, a CORBA-based distributed system. Builds on reconfiguration work in Polylith (Hofmeister and Purtilo, 1993). Like that work, the DRM passivates links between objects before reconfiguring them and transfers state to initialize new versions. The authors define formal consistency and efficiency constraints for reconfiguration, and argue that their reconfiguration algorithm is optimally efficient (i.e., causes minimal disruption).

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

Bloom's thesis describes reconfiguration in Argus, a strongly-typed distributed system composed of modules called "guardians." A guardian resides at a single node and is composed of a set of processes and a set of state variables. State can either be volatile or stable; stable state is guaranteed to survive crashes. A guardian's interface is a set of handlers (i.e., RPCs). Handlers are implemented using atomic actions (i.e., transactions) that guarantee consistency, even across multiple guardians at different nodes. Bloom's work addresses the problem of replacing implementations in this environment. The smallest unit of replacement is a single guardian. Subsystems, which are composed of multiple guardians, may also be replaced atomically. To support this, Bloom defines the interface of a subsystem as a subset of the handlers of the guardians in that subsystem. Bloom also defines a formal model to determine which replacements are legal; these are those replacements that preserve or invisibly extend the replaced subsystem's continuation abstraction. Bloom presents several examples of replacements that seem intuitively legal but that actually violate this condition. For example, replacing a unique ID generator may violate future behavior by repeating an ID. Extending an abstraction by adding a "delete" operation may break clients that depend on data existing forever (i.e., that only check for existence once). Bloom also argues that successive replacements can eventually restrict a continuation abstraction until no more replacements are possible. The actual mechanisms used to replace subsystems allow a user to manually locate, add, remove, and restart guardians; get, put, and optionally translate their state; and fetch and rebind handlers. A sequence of these actions compose a replacement transaction; this transaction either may wait until it can acquire exclusive lock on all required guardians or may preempt and abort other clients' transactions. Bloom does not detail an implementation and cites the need for a higher-level user interface to replacement.

[26] Toby Bloom and Mark Day. Reconfiguration in Argus. In Intl. Conf. on Configurable Dist. Systems [57], pages 176–187. Also in [55], pages 102–108.

Defines a correctness condition for reconfiguration: "continuation abstractions are preserved or invisibly extended by a replacement." That is, module (guardian) replacements must be backward (upward) compatible and must continue the behavior of the original module

(e.g., by transferring the old module's state). Reconfiguration quiesces the modules to be replaced; client transactions on those modules abort. The authors describe two kinds of upgrading infrastructures: system-supported replacement (SSR) and application-level replacement (ALR). SSR requires hooks in the Argus system to allow replacement and an additional indirection on handler calls. ALR requires participation by designers of modules and clients of modules to support upgrades.

[27] Philippe Breche, Fabrizio Ferrandina, and Martin Kuklok. Simulation of schema change using views. In *Database and Expert Systems Applications*, pages 247–258, 1995.

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

Discusses tradeoffs in the design of giant-scale services that allow for graceful degredation of service under load and failure and support for online evolution. Advocates automatic upgrade systems, and describes three approaches: fast reboot (everyone at once), rolling upgrade (round-robin), and big flip (partition the system, then upgrade each partition). Insists that these systems need a safe and fast way to roll back to the old version, since new versions tend to be buggy. Mentions that many systems use a staging area where the new software is set up alongside the old software before going live — makes switchover (in either direction) easy.

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

[30] M. R. V. Chaudron and F. van de Laar. An upgrade mechanism based on publish/subscribe interaction. In Workshop on Dependable On-line Upgrading of Dist. Systems [97].

Advocates the use of publish/subscribe as an interaction style for upgradeable component-based systems. Publish/subscribe induces less coupling than request-response (e.g., RPC), since components do not have to interact synchronously and do not have to know each other. Therefore, components can be replaced (more) easily; a configuration manager (CM) manages these replacements. Authors do not consider upgrades that require state transfer between components or upgrades to the CM.

[31] Injun Choi, Sungmoon Bae, Namchul Do, and Myungwhan Yun. Backward propagation of engineering constraints in active object-oriented databases. In *Proc. of 22nd International Conference on Computers and Industrial Engineering*, pages 20–23, Cairo, Egypt, December 1997.

Describes a use of triggers to maintain integrity constraints among a set of related objects. This is mildly related to how simulation objects must propagate mutations to objects further down the chain and deal with problems that occur when their constraints are violated.

[32] Jonathan E. Cook and Jeffery A. Dage. Highly reliable upgrading of components. In *Intl. Conf. on Software Engineering*, Los Angeles, CA, 1999.

Maintains and runs multiple versions of a component simultaneously to avoid introducing errors at upgrades. For example, suppose version 1 of a component has a method whose input is any nonnegative number, and suppose version 2 accepts any number. Then, this system uses version 2's output for nonpositive numbers and uses version 1's for positive numbers (the idea being version 1 probably works fine for positive numbers, but version 2 might be broken). The system monitors version 2's output on positive numbers and records statistics on whether it makes any errors. The upgrader can examine these statistics to determine whether version 1 can be removed in favor of version 2. This work does not seem to address the problem of state divergence between components (i.e., since version 2 sees some requests that version 1 does not, version 2's state may diverge from version 1's).

[33] Jonathan E. Cook and Navin Vedagiri. Reliable upgrading through multi-version execution. In Workshop on Dependable On-line Upgrading of Dist. Systems [97].

Allows multiple versions of a function to exist simultaneously, and uses an arbiter to dispatch calls dynamically to different versions of a function. Keeps the old version around to protect the system from errors introduced by a new version, until the new version is deemed safe.

[34] Virginia C. de Paula, G. R. Ribeiro Justo, and P. R. F. Cunha. Specifying dynamic distributed software architectures.

[35] P. Devanbu, M. Gertz, and S. Stubblebine. Security for automated, distributed configuration management. In *ICSE Workshop on Software Engineering over the Internet*, April 1999.

Identifies security issues for automatic software management and a research plan to address them. Integrity must be guaranteed for the software being shipped from vendor to user, the user's configuration, and messages from user to vendor that describe configurations. Authentication is needed to identify software vendors and licenced software users. Privacy protections are needed for software components (because of their intellectual property value) and for software configurations (because they may reveal sensitive data). Finally, delegation is needed, e.g., to let administrators delegate configuration control to vendors and to let vendors delegate configuration checking to a testing lab.

[36] Chryssa Dislis. Improving service availability via low-outage upgrades. In Workshop on Dependable On-line Upgrading of Dist. Systems [97].

Gives an industry perspective (Motorola, Inc.) on the importance of upgrading software without interrupting service, and suggests (common sense) techniques for minimizing downtime. For example, preparing the system for the upgrade, taking backups in case roll-back is needed, minimizing user interaction (since this is typically slower than automated actions and reduces error), etc. Note that the full paper is not available yet.

[37] Dominic Duggan. Type-based hot swapping of running modules. In *Intl. Conf. on Functional Programming*, pages 62–73, 2001.

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

Separates distributed systems into "zones" that upgrade using stop-the-world. Uses "change absorbers" and "transformers" to convert objects as they move from zone to zone.

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

[by Steven Richman] Fabry introduces many of the basic concepts of dynamic upgrades. He proposes the use of pointer indirection to resolve code and data references to current versions. Data structures are upgraded lazily and can have their representations transformed arbitrarily during an upgrade. Because upgrades can introduce incompatibilities between code and data of different versions, data structures are tagged with version numbers. These numbers are checked before the execution of data access routines: if a data structure's version is not what a routine expects, then the out of date data or code is upgraded. Fabry makes no mention of quiescence or valid times at which upgrades can be safely applied; instead, he relies upon synchronization on data structure locks to control the progress and ordering of an upgrade's modifications. This scheme requires the restart of code segments that are older than the data they are trying to access, and therefore seems to not work if a side effect precedes a data structure access in a flow of execution.

[40] Ophir Frieder and Mark E. Segal. Dynamic program updating in a distributed computer system. In *IEEE Conf. on Software Maintenance*, pages 198–203, Phoenix, AZ, October 1988.

[41] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, pages 111–128, February 1991.

Describes PODUS (see Segal and Frieder, 1993).

[42] Sanjay Ghemawat. Google, Inc., personal communication, 2002.

[43] Stephen Gilmore, Dilsun Kirli, and Chris Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, University of Edinburgh, December 1997.

[44] R. Govindan, C. Alaettino, and D. Estrin. A framework for active distributed services. Technical Report 98-669, ISI-USC, 1998.

An active distributed service (ADS) is composed of cooperating agents located on nodes on a network. Agents are extensible by plugging in new event handlers (e.g., message handlers). Handlers are pushed from agent to agent or are retrieved on demand. Each physical node has one "actuator" that sends and receives handlers (cf. upgrade dissemination). Each ADS agent on a node has an "envoy" that uses the actuator to get handlers. Together, the actuator and the envoys compose the ADS's runtime "substrate".

[45] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software Practice and Experience*, 23(9):949–964, September 1993.

[by Steven Richman] Gupta and Jalote take a rather different approach to on-the-fly upgrades. Instead of modifying a running process, they suspend the process and copy it to a new, upgraded process. The copy process has a new code segment that reflects the update, and data and stack segments that are duplicated from the old processes (with pointers modified as necessary). A runtime library contains special versions of the open and close system calls that allow open file descriptors to be transfered to the new processes. The system only considers an upgrade valid if none of the modified functions are being executed at the time the update occurs. This is enforced by having the application run as a child process of an upgrader process: the upgrader process uses the ptrace debugging system call to monitor the application's stack. This quiescence requirement precludes the update of long-lived functions that are often or always on the stack, such as top level functions or event processing loops. Gupta and Jalote posit that these sorts of functions generally can be made static, with the "real work" delegated to shorter-lived functions. It is not obvious that this is true, and their quiescence requirement may prove onerous in many applications. The programmer is allowed to specify a single state transformer function that is executed when the rest of the upgrade is complete.

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

Uses servers called "release docks" at software producers and "field docks" at software consumers to disseminate new software and updates. Users download and authorize installation agents which in turn install the requested software and install update agents. A global event system routes update notifications to update agents, which in turn download and install updates. A hierarchical registry system standardizes descriptions of software packages, their file structure, and their interdependencies. If an installation or update depends on another package, that package is automatically installed or updated as needed. Local agents enforce access control, e.g., by mapping registry changes onto the file system.

[47] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A cooperative approach to support software deployment using the Software Dock. In *Intl. Conf. on Software Engineering*, pages 174–183, 1999.

[48] Steffen Hauptmann and Josef Wasel. On-line maintenance with on-the-fly software replacement. In Intl. Conf. on Configurable Dist. Systems [59], pages 70–80.

Supports replacement of modules, called "actors", in the Chorus real-time OS. Actors can contain several threads and provide an interface composed of several communication ports. The replacement approach does not depend on specially-written apps; instead, app code is modified to make the app replaceable. The authors claim that most of this modification can be done automatically. Modifications include (1) adding a thread to run the replacement, (2) adding state capture/restore functions at exchange points (places where threads can block), (3) addings handlers for aborted system calls, (4) adding instructions to restore the call stack after replacement.

[49] 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.

Used by several reconfiguration systems to transfer state from old versions of components to new ones.

[50] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–23, 2001.

[by Steven Richman] Hicks et al. implement a dynamic update system in a C-like imperative language. Their system allows for data transformation during an upgrade. An upgrade consists of dynamically-linked code and optional state transformer functions. An upgrade's type safety is guaranteed with a proof-carrying typed assembly language (it is unclear why the type analysis cannot be carried out at compile time). New code is attached to old code by relinking references. A tool automatically generates simple state transformer functions based on code changes, minimizing programmer work. The programmer is required to specify a single quiescent point in the application at which upgrades can safely occur, and this point cannot change across versions. An upgrade happens atomically at the specified time. The authors apply their upgrade system to a single-threaded event-driven web server that is amenable to quiescence identification, but it is not clear that update timing can be specified easily in multithreaded or more complex applications. In general, update timing is an important and difficult problem in systems that seek to upgrade running applications mid-execution.

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

[by Steven Richman] Hjalmtysson and Gray's dynamic C++ classes represent an upgrade system similar in spirit to but simpler than Fabry's scheme. They permit updates at class granularity by providing a library with a generic template class that serves as a proxy for indirect access to dynamic classes. Dynamic linking intromits new code. Dynamic C++ classes avoid the problem of upgrade completeness and quiescence by allowing objects with old class versions to persist until they are destroyed; an upgrade applies only to new objects. Objects of different versions can coexist because the system forces dynamic classes to inherit from abstract interfaces that cannot change across versions. This constraint necessarily limits program evolution. Further, the policy of keeping objects with out of date class versions until deletion is ill-suited to applications with long-lived objects: an upgrade is not complete until all objects from the old version have been destroyed, and this may never occur. This becomes problematic if a critical bug fix must be applied to a long-lived object. Ultimately, the onus is placed on the programmer to delete and reconstruct objects that would not otherwise be updated. It is clear, then, that automatic transformation of live objects is a desirable property

in an upgrade system. Hjalmtysson and Gray's method has several strengths, though; namely, it is an efficient implementation of dynamic updates in a modern programming language and uses only those features already present in the language and linking environment–no language extensions or special runtime systems are required.

[52] C. Hofmeister, E. White, and J. Purtilo. Surgeon: A packager for dynamically reconfigurable distributed applications. In Intl. Conf. on Configurable Dist. Systems [57], pages 164–175. Also in [55], pages 95–101.

Describes a way to package an upgrade to software components in a distributed system. Packaging analyzes interface bindings, determines how components should be integrated, generates interface software to connect components, and creates configuration commands to build the application. A "catalyst" module on each node actually runs reconfigurations using a package (cf. the UL). Packaging may require component participation to save the state, transform state, restart the component, or delay upgrades until a suitable point (so that consistency can be maintained). The authors categorize the kinds of components that can be reconfigured without any such participation: these are those modules that neither require state transfer, nor special initialization, nor synchronization with other modules.

[53] Christine R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, University of Maryland, College Park, 1994. Also available as Technical Report CS-TR-3210.

Hofmeister's thesis on her 1993 work with Purtilo.

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

Inspired by Kramer and Magee's reconfiguration model in Conic, this work adapts that model to the Polylith distributed environment (supports general message-passing, not just RPC). Reconfigurations can change module implementations, application structure (e.g. add and remove modules), and application geometry (e.g. physical location of modules). This work extends the Conic work by adding support for capturing and restoring process state via an intermediate abstract representation [49]. Reconfiguring modules must be quiescent; messages received during the reconfiguration are buffered and are used to initialize the replacement module.

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

[56] INSERT: Incremental software evolution for real-time applications, 1997.

This entry is a placeholder until a suitable publication is found. From the web page: Our objective is the development of a capability package that will permit safe on-line upgrading of hardware and software in spite of residual errors in the new components. This package will facilitate a paradigm shift from static design and extensive testing to safe upgrades of real-time safety critical systems. The package will be implemented and demonstrated in the Lockheed Martin flight simulation hotbench.

[57] *Intl. Workshop on Configurable Dist. Systems*, London, England, March 1992.

[58] *2nd Intl. Workshop on Configurable Dist. Systems*, Pittsburgh, PA, March 1994.

[59] *3rd Intl. Conf. on Configurable Dist. Systems*, Annapolis, MD, May 1996.

[60] *4th Intl. Conf. on Configurable Dist. Systems*, Annapolis, MD, May 1998.

[61] R. H. Katz, M. Anwarrudin, and E. Chang. A version server for computer-aided design data. In *Proc. 23rd Design Automation Conference*, pages 27–33, 1986.

Describes a way to keep a history of object versions by allowing later versions to keep a pointer to earlier ones. Need to read this to determine whether or not this is similar to SOs. (I don't think so)

[62] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(4):424–436, April 1985.

Describes how to upgrade a distributed system that is specified in Conic, i.e., as a set of modules and connections between them. Messages between modules may be one-way (asynchronous) or request-response (RPC-like). Upgrades are specified as declarative change commands (link, unlink, create, delete, etc); a Configuration Manager (CM) translates these into operating system commands and executes them. Declarative commands allow the CM to select the best "change strategy," e.g., to minimize downtime (cf. upgrade schedule). The CM checks that any changes obey module interface type signatures. The upgrader must specify change commands directly; the system does not attempt to infer change commands from before and after versions of the configuration (e.g., Hicks, 2001 and Tewksbury, 2001). The CM itself is specified in Conic, but is not upgradable. System does not quiesce modules, and so cannot guarantee state consistency. System does not support state transfer or translation between versions. System does not provide atomic upgrades, i.e., an upgrade may fail and leave the system in an inconsistent state.

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

Argues for a separation of functional application concerns from structural configuration concerns. Distributed applications are described as interconnected components. Configuration changes are specified declaratively; a configuration management system translates this specification into a "change transaction" that can create, remove, link, or unlink components. To preserve application consistency, components must be able to become "passive," i.e., stop initiating but continue serving transactions. Given this capability, the configuration management system can passivate the appropriate components before actually changing the system structure. A component can be removed if it is "quiescent", i.e., it is passive and all components linked to it are passive. The authors mention that an alternative to quiescence is to use recovery to restore application consistency, but they argue that this complicates applications (but they also mention recovery may be necessary to deal with failure). The initial presentation assumes transactions are independent (i.e. not nested), but later sections relax this assumption (either by passivating dependent components or by aborting the dependent transactions). The system requires that the initiator of a transaction be informed of when a transaction completes (i.e., no one-way messages). The system can be extended to support multiple, concurrent change transactions by passivating more components. The system does not support state transfer between old and new components, but the authors mention that this is possible between quiescent components. In future work, the authors mention that an application could minimize system disruption by instigating change when quiescence is detected, rather than externally imposed.

[64] J. Kramer, J. Magee, and A. Young. Towards unifying fault and change management. In *IEEE Workshop on Future Trends of Dist. Computing Systems in the '90s*, pages 57–63, Cairo, 1990.

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

Excellent related work on schema evolution systems. Focus is on identifying (inferring) steps in schema evolution from before and after definitions (i.e., does an intelligent diff to infer refactorings). Also generates derivation functions to initialize new state from old values (i.e., state transform functions).

[66] Barbara Liskov. Software upgrades in distributed systems, October 2001. Keynote address at the 18th ACM Symposium on Operating Systems Principles.

[67] Chang Liu and Debra J. Richardson. Using RAIC for dependable on-line upgrading of distributed systems. In Workshop on Dependable On-line Upgrading of Dist. Systems [97].

A RAIC controller provides a static interface to an array of similar or identical components. The RAIC automatically handles failover and recovery of components. Thus, RAIC can support component updates by allowing new versions to be added to the component array. If the new version of a component is faulty (e.g., raises an exception), RAIC intercepts the fault and redirects the call to an older version of the component. Furthermore, different callers can use different versions of the components. It is not clear whether this system avoids state divergence between components of different versions.

[68] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *European Conf. on Object-Oriented Programming*, 2000.

[by Steven Richman] Malabarba et al. bring dynamic classes to Java by modifying the virtual machine and adding a dynamic class loader. Type correctness is checked when an upgrade in compiled. Specifically, their compiler guarantees that the set of new and changed classes comprising an update forms a complete upgrade if the changes are applied atomically. In their implementation, upgrades occur in an atomic global update, but objects are transformed lazily, as in our persistent object base. A quick initial marking phase tags all reachable objects that need to be upgraded, and any subsequent references to marked objects trap to an upgrader that suspends all threads and brings the objects up to date. No general state transformation facility is provided; fields are simply copied from old objects to new objects and new fields are initialized to default values.

[69] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 287–296, Helsinki, Finland, September 2003.

[70] 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.

Uses gateways between heterogeneous distributed platforms to provide "federation transparency" (cf. "evolution transparency"). When using objects from a different domain, they look like objects in the local domain.

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

Presents a model for class (schema) versioning in OODBs. Schema versioning is different from schema modification: modification has a single logical schema that is updated, e.g., by changing a class definition. All instances of the class are eventually (eagerly or lazily) updated. In versioning, every change to a class results in a new version. Each instance is created using a given version. This version never changes, i.e., an instance never transforms to a new version. The author's model uses "update" and "backdate" methods to convert method/field accesses up and down versions. This is more flexible than ENCORE [87, 99] because it allows accesses to have different semantics in each version. This system also stores old values that are removed in later versions.

[72] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus: An architecture for extensible distributed systems. In *14th ACM Symposium on Operating System Principals*, Asheville, NC, 1993.

A communication medium that supports the transfer of objects using either publish-subscribe or RMI. Objects are identified using hierarchical (DNS-like) names called "subjects." Clients get data by subscribing to subjects. Servers name published data using subjects. Clients locate services (e.g., for RMI) by publishing interests. Thus, new servers can be introduced without altering clients. Data is disseminated using ethernet broadcast on LANs and an overlay network in the wide area. Objects are dynamically typed, so new types can be introduced (although clients can use the new types only if they know how to handle a supertype). The system supports legacy applications and services using "adapters" to translate between the app domain and the Information Bus object format.

[73] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Intl. Conf. on Software Engineering*, Kyoto, Japan, April 1998.

Supports runtime software evolution by adjusting "connectors" between components dynamically. Connectors are themselves components that regulate communication and abstract the underlying mechanisms. Uses imperative commands (add, link, start) to direct reconfiguration.

[74] Vivek Pai et al. CoDeeN.

A CDN deployed on PlanetLab that upgrades its nodes about twice a week, causing only about 20 seconds of downtime per node. New versions are simply scp'd to the nodes and the software is restarted to use the new code. Versions are typically backwards-compatible; on the rare occasions when a new version is incompatible, version numbers are used to distinguish new calls from old ones (which are rejected).

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

[76] Felix Rauch, Christian Kurmann, and Thomas M. Stricker. Partition repositories for partition cloning—OS independent software maintenance in large clusters of PCs. In *Proceedings of the IEEE International Conference on Cluster Computing 2000*, Chemnitz, Germany, November 2000.

[rauch] In this paper we looked at the problem of minimizing the amount of data to archive (or upgrade) multiple versions of installations in a multi-use cluster.

[77] Felix Rauch, Christian Kurmann, and Thomas M. Stricker. Optimizing the distribution of large data sets in theory and practice. *Concurrency and Computation: Practice and Experience*, 14(3):165–181, April 2002.

[rauch] The problem of installing multiple operating systems on our 128-node cluster motivated us to do this study on the distribution of large data sets (basically hard-disk partitions).

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

Extends service selection to choose best match via "service distance" (best-fit specification matching)

[79] Joel Richardson and Peter Schwarz. Aspects: Extending objects to support multiple, independent roles. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, volume 20, pages 298–307, May 1991.

Describes how to integrate aspects (wrappers) into the type system, so that a single object may evolve over time with new state and new behavior. Similar to SOs, since aspects need not implement a subtype of the base object's type and may contain their own state. But this paper does not discuss what happens when the base object enters a state that doesn't make sense in the aspect.

[80] Tobias Ritzau and Jesper Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, May 2000.

JDrums: Uses lazy upgrades to convert classes and objects to new versions. An upgrade consists of a class converter that converts static class data and an object converter that converts instances. The system uses a modified JVM to keeps old versions of classes and objects around so that old references continue to work. Does not comment on the problem of state divergence between different versions of the same object. Uses Jini to deploy upgrades in distributed systems, but has no mechanism to synchronize or otherwise schedule distributed upgrades. Converter routines cannot call methods of old or new objects – they can only copy and convert object state.

[81] Mark E. Segal and Ophir Frieder. Dynamically updating distributed software: supporting change in uncertain and mistrustful environments. In *IEEE Conf. on Software Maintenance*, pages 254–261, October 1989.

[82] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2), March 1993.

A good review of several updating systems. Categorizes software-based dynamic updating systems as those that replace abstract data types (e.g., Fabry), replace servers in client-server systems (e.g., Argus), update in constrained message-passing systems (e.g. Conic), and update programs in procedural languages (e.g., PODUS). Also details PODUS: a procedure-oriented dynamic updating system that can replace a procedure definition provided it is inactive, i.e., not on the stack, not used by any proc on the stack, and not semantically depended on by any proc on the stack. PODUS works in distributed environments that use RPC. Semantic dependencies between procs are specified by a programmer; semantically dependent procs must reside at the same physical site. PODUS uses "interprocedures" to map calls from an old version of a proc to a new version and uses "mapper procedures" to copy/convert static state from one proc to another. Thus, PODUS can support multiple interacting versions of concurrent, distributed programs.

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

Describes how to use mappers to enable cross-version interoperation during distributed upgrades. Argues that this interrupts service less than systems that atomically upgrade all clients and servers that use a changing interface. Describes a UI that guides the evolver (upgrader) through automatic generation of mapper (simulation) code. Restricts autogenerated wrappers to 1-to-1 mapping from old method calls to new ones (in general, the wrapper could call multiple methods on many different servers). Categorizes the supported kinds of interface evolution, including subtyping. Also discusses chains of mappers, backwards mappers (new-to-old), propagation of supertype changes to subtypes, and optimizations (like deprecating old mappers).

[84] Twittie Senivongse and Ian Utting. A model for evolution of services in distributed systems. In Spaniol Schill, Mittasch and Popien, editors, *Distributed Platforms*, pages 373–385. Chapman and Hall, January 1996.

Uses mapping operators to provide "evolution transparency" (behavioral compatibility between versions). No forward compatibility, SO state, failure mode, or correctness criteria. Supports adapter chaining, automatic adapter generation, and non-subtype evolution.

[85] Lui Sha, Ragunathan Rajkuman, and Michael Gagliardi. Evolving dependable real-time systems. Technical Report CMS/SEI-95-TR-005, CMU, 1995.

Describes the Simplex architecture for supporting evolution of real-time systems that use commercial off-the-shelf (COTS) components. Upgrades are supported by grouping a set of analytically redundant components (i.e., that satisfy the same abstract spec) into a subsystem module. Each module contains a safety component that is assumed correct but may be inefficient, a baseline component that acts as the "leader" of the replica group, and an optional new component that is evaluated against the other two. Each module also contains a management system that monitors the components for errors (e.g. functional or resource utilization). If the new component behaves correctly according toa user-specified metric, the system replaces the baseline component with the new one. A two-phase protocol is used to atomically switch over a set of distributed components.

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

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

This system (ENCORE) makes type changes by creating new versions. Objects (instances) retain their version unless they are coerced (transformed) to another version. To satisfy cross-version method calls/field accesses, each type has a version set interface (VSI). The VSI is the union of all methods anf fields of all the versions of that type; therefore, any program that accesses any version of the type can access the VSI correctly. Accesses are passed from the VSI to the underlying instance. If the access fails (e.g., because the target method/field doesn't exist or the access would read/write an illegal value), then an error handler in the VSI can substitute a response (e.g., default/alternate return value). This system handles not only changes to a type definition, but also changes to the type hierarchy. It also uses boolean formulas in type specs to automatically determine when handlers are needed. This system does not support changes to method/field semantics across versions (e.g., readOdometer() cannot change from returning miles to km; instead, you need to remove one method and add another). More importantly, the number of handlers scales as the square of the number of versions, because each time a version is added, handlers must be added for all the other versions.

[88] Marcin Solarski and Hein Meling. Towards upgrading actively replicated servers on-the-fly. In Workshop on Dependable On-line Upgrading of Dist. Systems [97].

Upgrades a replicated server by upgrading each replica in turn. Assumes that (1) multiple upgrades do not interleave, (2) version v+1 offers a compatible interface to version v, (3) there exists a mapping from version v's state to version v+1, and (4) clients only use extensions offered by v+1's interface after all replicas are upgraded. Upgrades are scheduled by totally ordering the replicas (using some replica identifier). Assumes full compatibility between different versions, so doesn't need to convert messages between versions (i.e., no simulation mode).

[89] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proc. of the Usenix Technical Conference*, 2003.

Implements dynamic interposition and hot-swapping for components in the K42 operating system. In the common case, nothing is interposed; interposers installed dynamically by modifying a call indirection table. Interposition can add wrappers to a component that can take action before and after each call to the component, like a profiler. Interposition also enables hot-swapping: an interposed Mediator blocks new calls to the component, lets the old calls drain, transfers state to the new component, then unblocks the calls. This scheme depends on the fact that requests to a component are short-lived, and this kind of component change is called Read-Copy Update (RCU).

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

Uses replication of CORBA objects to support upgrades without interrupting service. Rather than allowing the system to exist in a hybrid state (i.e., where different objects are at different versions), the upgrade executes an "atomic switchover" that changes all objects from one version to another. Reliable, totally-ordered multicast ensures atomicity. If the upgrade changes an objects' interface, all clients that use that interface must also be upgraded at the switchover. Furthermore, all affected objects must be quiescent when the switchover occurs. The system uses wrapper functions to loosen this quiescence requirement (i.e., by translating old calls to new ones). The system provides an "upgrade preparer" tool that automatically generates wrapper functions and state transformers given the old and new versions of an object's code. This work follows that of Kramer and Magee, Hofmeister and Purtilo, and Bidan et al.

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

[92] E. Truyen, W. Joosen, and P. Verbaeten. Consistency management in the presence of simultaneous client-specific views. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 501–510. IEEE Computer Society, October3–6 2002.

[93] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Nørregaard Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01)*, pages 233–242. IEEE Computer Society, May12–19 2001.

[94] Robert K. Weiler. Automatic upgrades: A hands-on process. *Information Week*, March 2002.

Weiler argues that fully automatic upgrades are dangerous, since automatic upgrade mechanisms often introduce incompatibilities in the system. Part of the problem is that each vendor develops and deploys their own automatic upgrade system, and systems from different vendors don't work together (e.g., to detect and resolve dependencies and incompatibilites). Standards for automatic upgrade systems would help address these problems by allowing different vendors' systems to interoperate. Another thing that could help is a log of all changes made to a PC's configuration and a record of when problems occur. This helps users identify bad upgrades; these upgrades can be removed if they support "automatic restoration" (i.e., rollback).

[95] Linda Wills et al. An open platform for reconfigurable control. *IEEE Control Systems Magazine*, June 2001.

Uses real-time CORBA to implement component-based control systems, and uses a publish-subscribe communication bus to loosen component coupling (and make reconfiguration easier). Components specify input and output ports along qith QoS constraints (priority, sample rate, execution time). Reconfiguration can create components, change port connections, and

alter QoS constraints. Authors propose to use common controller design patterns to support common reconfigurations. Cites Oreizy et al. as main previous work.

[96] Eric Wohlstadter, Brian Toone, and Prem Devanbu. A framework for flexible evolution in distributed heterogeneous systems. In *International Workshop on Principles of Software Evolution*, pages 39–42, Orlando, Florida, 2002.

[97] *Workshop on Dependable On-line Upgrading of Dist. Systems in conjunction with COMPSAC 2002*, Oxford, England, August 2002.

[98] Robert Wrembel. Object-oriented views: Virtues and limitations. In *13th International Symposium on Computer and Information Sciences (ISCIS)*, Antalya, November 1998.

A survey of techniques for supporting object-oriented views. Relevant to upgrades because simulation objects for different versions act like different views on the state of a node.

[99] Stanley B. Zdonik. Maintaining consistency in a database with changing types. *SIGPLAN Notices*, 21(10):120–127, October 1986.

A nice summary of the ideas presented in [87]. Ignores the issue of changes to the type hierarchy; just deals with version changes that add/remove/change methods and fields.