Rolling Upgrades for Continuous Services

Antoni Wolski and Kyösti Laiho

Solid Information Technology, Merimiehenkatu 36D,
FIN-00150 Helsinki, Finland
{antoni.wolski, kyosti.laiho}@solidtech.com

Abstract. With the advent of highly available systems, a new challenge has appeared in the form of the requirement for *rolling upgrade* support. A rolling upgrade is an upgrade of a software version, performed without a noticeable down-time or other disruption of service. Highly available systems were originally conceived to cope with hardware and software failures. Upgrading the software, while the same software is running, is a different matter and it is not trivial, given possible complex dependencies among different software and data entities. This paper addresses the needs for rolling upgradeability of various levels of software running in high-availability (HA) frameworks like the Availability Management Framework (AMF) as specified by SA Forum. The mechanism of a controlled switchover available in HA frameworks is beneficial for rolling upgrades and allows for almost instantaneous replacement of a software instance with a new version thereof. However, problems emerge when the new version exposes dependencies on other upgrades. Such dependencies may result from new or changed communications protocols, changed interfaces of other entities or dependency on new data produced by another entity. The main contribution of this paper is a method to capture the code, data and schema dependencies of a data-bound application system by way a directed graph called Upgrade Food Chain (UFC). By using UFC, the correct upgrade order of various entities may be established. Requirements and scenarios for upgrades of different layers of software including applications, database schemata, DBMS software and framework software are also separately discussed. The presented methods and guidelines may be effectively used in designing HA systems capable of rolling upgrades.

1 Introduction

The concept of *service continuity* embraced in the goals of the Service Availability Forum[1] is based on the notion that very short breaks in operation of service-providing applications are tolerable to a certain extent. This extent is specified using the availability measure *A* (percentage of the time a service is operational, as related to the total time the service is supposed to be operational) and, possibly, a maximum duration of a break (equal to mean time to repair, MTTR) or a frequency of breaks (represented with mean time between failures, MTTF). The three quantities are bound together with the formula:

[1] www.saforum.org

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp.175–189, 2005.

$$A = \frac{MTBF}{MTBF + MTTR} \bullet 100\%$$

In the view of high availability standards like those of SA Forum, the main culprits preying on service continuity are failures—both of hardware and software. To deal with them, the system embodies redundancy both in hardware and software, managed by a high availability framework like AMF (Availability Management Framework) [4] of SA Forum.

According to the SA Forum AIS (Application Interface Specification) model [1], redundancy is maintained at the level of *service units* that may comprise of one or more components. In the simplest redundancy model, called 2N, the two units, *active* and *standby* make up a *mated pair*, and the redundant application components are organized in pairs in the corresponding units. Should a failure occur, the failed active (service) unit (hardware or software) is quickly replaced with a corresponding standby (service) unit. This operation is called a *failover*. Switching of the roles of units may be done also on request, in a no-failure situation, and this we will call a *switchover*. Switchovers are useful in various maintenance situations as will be seen in the sequel. Service continuity is preserved if, in the presence of failures, the required service availability level is maintained. If a standby system fails, it is repaired and brought back into synchrony with the active unit. Such a failure does not normally cause an interruption to the service.

All systems face a need for component replacement and upgrades from time to time. The need to facilitate software upgrades is demanding because a system with continuous service uptime expectation can not be just stopped for maintenance and upgrade. In order to provide service continuity, the hardware and software upgrades have to be performed on a running system in such a way that the availability requirements are met. We will call such upgrades *rolling upgrades*.

The concept of the rolling upgrade incorporates the notion of using the standby units present in a HA system and thus may be considered a special case of a *dynamic upgrade* in general [11]. It should be noted, hover, that engaging standby units in the upgrade process may temporarily jeopardize the availability level of the overall service because it may happen that the standby unit may not be available for failover, should this be needed. For this reason, we propose to use *spare units*, in place of standby units, whenever the service availability in endangered. Spare units are units that are not assigned any active or standby role. Such units are available in many HA system platforms. The choice whether to use the spare unit or not depends on the anticipated upgrade duration and the criticality of the component being upgraded. In the update scenario examples presented below, we make some educated decisions about using the spare units. In reality, such decisions have to be made on the basis of more accurate information about the required availability level and the duration of the upgrade.

Given the complexity of modern telecommunications systems where implementations are becoming increasingly software-driven, several interrelated software layers have to be recognized. In this paper, we are concentrating on systems utilizing database-centric applications, and thus the software layers considered for rolling upgrades are:

- Operating system and availability framework
- Database management system
- Database schema
- Database applications

The above layers are schematically shown in Fig. 1, together with the relevant interfaces among them.

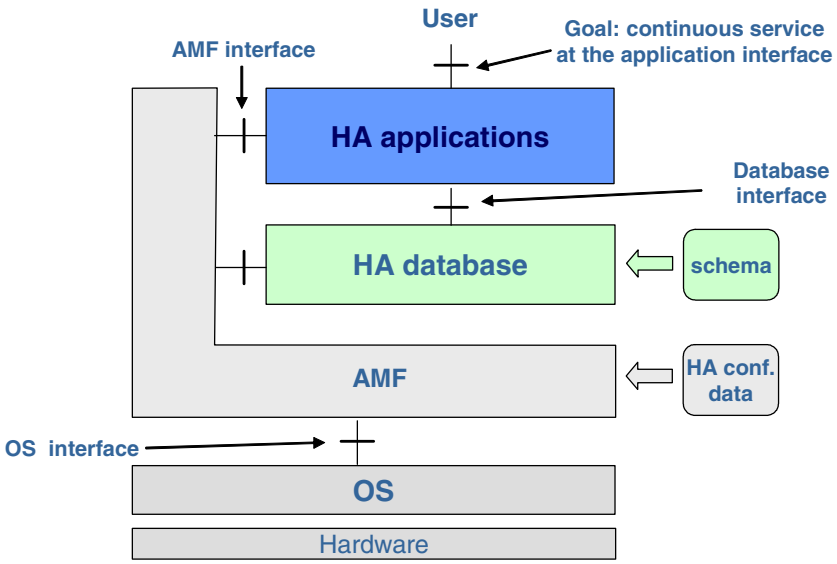

Fig. 1. Layers of software in an HA system

In Section 2, we survey the related work. In Section 3, the Upgrade Food Chain diagram is introduced with the purpose of capturing upgrade dependencies. In Section 4, requirements and scenarios associated with upgrades at different software layers are discussed. We conclude by summarizing the methods and guidelines produced.

2 Related Work

From the outset of uninterruptible systems, the needs for evolutionary changes, in a running system, have been recognized [7]. Consequently, various methods of dynamic (or live) upgrading (or updating) have been proposed (for review of early dynamic upgrading systems, see [11]). Researchers strived for achieving automatic upgrading systems and thus the proposed methods dealt with homogeneous components of low granularity. The update granules were abstract data types in Argus [2], procedures in PODUS [11] and tasks (or transactions) in Conic [7]. The emergence of well-defined component-based frameworks, like CORBA, J2EE and .NET, has offered new opportunities because of the unified component management and a possibility to represent component metadata in a natural way. There are methods for dynamic upgrading of CORBA components [14][8], Java RMI servers [12] and methods adaptable to J2EE EJB components [3]. Following the generally perceived needs, OMG has started an effort to produce the CORBA online upgrade specification [9], too.

Traditionally, the dynamic upgrades are expected to be *unattented* (i.e. automatic) and *safe* [3], i.e. not disrupting other components of the system. When building such a system, one has to answer two questions:

1) How to obtain and represent the necessary change and dependency information (upgrade metadata)?
2) How to execute the upgrade?

It is easier to answer the latter question once there is a satisfactory answer to the former one. Efforts have been made to extract the necessary metadata from the component interface specifications [14]. However, as the authors of [11] point out: "[fully automatic dynamic updating] cannot work properly if semantic information is needed to perform any aspect of the updating". Consequently, human input is needed to provide some of the metadata. An example is the ENT model (ENT stands for: Exports, Needs, Tags) [3] where the interface metadata is annotated with the changes in *provided-requested* relationships among components. Once the sufficient amount of metadata is produced, it can be used in unattended upgrading.

Inter-component dependency diagrams were introduced in [7]. In our work, we go further by introducing the Upgrade Food Chain (UFC) diagram that captures the version-specific change information only. This does not mean that the full dependency information is not needed: the change information is obtained by way of the differential analysis of the full dependency information.

A requirement for the component to be quiescent before it can be upgraded is often presented [14]. However, we argue that, in the presence of an HA framework like AMF, the components need not be necessary quiesced because they are not quiesced when a failover happens.

Similarly, it is required that the internal state is passed over to the new version of the component, to preserve the component correctness [11]. Our position is that we do not have to take care of that because the inherent nature of an HA component incorporates the notion of preserving the state in the presence of failover (or switchover). The means for achieving the preservation of state are application checkpoints [4] and writing the state into an HA database [5].

We are not aware of any work related to dynamic upgrades in large and diversified systems lacking a common component framework. In this work, we utilize the HA characteristics of a system, to ease the implementation of dynamic upgrades.

3 Rolling Upgrades: Dependencies and Requirements

3.1 Dependency Types

A major problem in facilitating rolling upgrades is that components of a system are interrelated. To picture the dependencies among system components, we choose to represent three different types of software components: *executable code* (standalone or library), *data* and *metadata*. *Code* represents independently startable applications and subsystems, and libraries to which they are linked. *Data* represents application data stored in a database or other persistent or run-time storage. Metadata means database schema declarations, such as table/data structures and integrity rules. One application version is typically bound to one version of schema, and may not work properly with a changed schema.

We introduce the Upgrade Food Chain (UFC) diagram to picture the dependencies among the software components discussed above. A possible UFC diagram may have the form shown in Fig. 2.

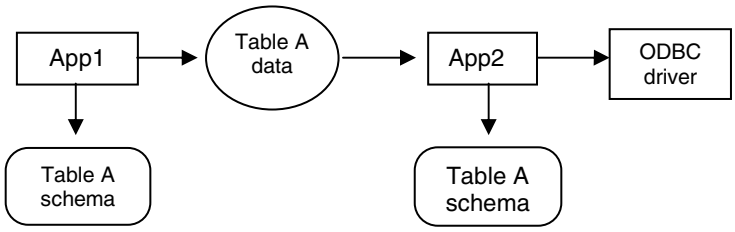

Fig. 2. Example UFC (Upgrade Food Chain) diagram

Consider a situation where two applications, *App1* and *App2* are upgraded to version *x+1*. *App1* uses data stored in a new table A. It thus needs also an upgraded database schema incorporating table *A*. The data in table *A* used by *App1* is produced by the upgraded *App2*. Additionally, *App2* needs a new version of an ODBC driver to function properly. The dependencies shown in the diagram are upgrade dependencies. Upgrade dependencies are special cases of inter-component function dependencies, as explained below.

Definition: Function Dependency
A component *A* is said to be *function-dependent* on component *B* if it requires some services or characteristics of component *B* to function properly.

If component *A* uses services of component *B*, it is said to be a *consumer* of services produced by *B*. Function dependencies among components are usually static and version-invariant. The reason is that, from the time of the component inception, its purpose and nature implies the related function dependencies. For example, all database-bound applications are function-dependent on the database schema, by definition. Exceptions from this rule may happen if the functionality of a component is changed significantly.

Knowledge of function dependencies is a sufficient, but not necessary, condition for execution of a safe multicomponent upgrade. Given an existing version *x* and the target version *x+1*, the necessary condition is the knowledge of version-specific function dependency, called upgrade dependency.

Definition: Upgrade Dependency
Assume we are upgrading components *A* and *B* from version *x* to *x+1*. Component *A* is *upgrade-dependent* on component *B* if the upgraded component *A* requires the functionality or characteristics increment, introduced in the upgrade of *B*, to function properly.

One can see that the purpose of upgrade dependability is to represent new dependencies that are introduced with a new version. If the two components involved are versioned in a different way, both new versions should be indicated in the dependency. On the other hand, if the function dependency of one component on another has not changed or is disappearing, with a given upgrade, it is not considered to be an upgrade dependency.

Definition: Upgrade Food Chain (UFC) Diagram
Upgrade Food Chain diagram is a directed graph, with each nodes being an instance of one of the three component types (code, data and metadata), and edges pointing to upgrade-dependent components.

Intuitively, the components should be upgraded in the reverse order of directed edges, starting from outmost components. All the components captured in a single UFC are considered a part of an *upgrade suite*. Upgrading of components in an upgrade suite has to be coordinated (ordered) so that the components can function properly during the upgrade process.

3.2 Assumptions About the System

Upgrade Granularity. The upgrade granularity we consider for SA-aware software is between (and including) the component and the service unit. A component is the smallest entity recognized by the AMF and also a natural unit of software development. A service unit (that comprises of components) is a unit of redundancy and thus switchovers are performed at this level.

Because both the concepts are irrelevant at the level of the operating system and the HA framework, the upgrade granularity for both is that a of a (cluster) node.

Distribution. An HA system is inherently distributed, not the least because of the hardware redundancy. Besides, the AMF has been planned for multi-computer clusters. Otherwise than assuming that components of one unit are co-located on the same cluster node, we do not make any references to the distributed nature of the system. We assume the function dependencies among components do not depend on the fact whether the components are co-located on a node or not.

Upgrade Transparency. When switchovers happen, the related component network addresses (service access points) change at each switchover. Upgrade transparency means that the consumer of the service, that is not upgrade-dependent on the upgraded service, should not be affected in any way by the upgrade. Because the upgrades we discuss are based on switchovers, the means for achieving upgrade transparency are the same as the means for achieving failure transparency, in an HA system, and we do not discuss it any further.

3.3 Trivial Upgrade: Independent Component

If a component upgrade is not dependent on any other component upgrade, it can be upgraded on its own because its upgrade suite does not comprise any other components.

To upgrade an independent component, a plain switchover may be applied. In this case, the procedure is shown below, given App_a^n and App_s^n are application instances of version n running as components in active and standby units, respectively.

To upgrade an independent code component App from version x to version x+1:

1) Stop the component App_s^x in the standby unit.
2) Install the new version of the component in the standby unit.
3) Restart the component as App_s^{x+1}.
4) Perform controlled switchover of units (App_a^x becomes App_s^x)
5) Stop App_s^x in the new standby unit.

6) Install the new version of the component in the standby unit.
7) Restart the component as App_s^{x+1}.
8) (Optional) Perform one more switchover if the original assignment of active and standby units was a preferable one.

Requirements. After performing step 4, the instances App_a^{x+1} and App_s^{x} have to interwork as a mated pair. If the active/standby operation at the component level involves communications between the active and standby component (e.g. to perform application state checkpoints), care should be taken of the need of the new version App_a^{x+1} to be able to communicate with the old version App_s^{x}, and possibly vice versa. If there is no intra-pair communications, e.g. if the component instances exchange data via a database, this concern is irrelevant.

Note that between steps 2 and 7, the system is vulnerable because it is running in stand-alone mode: there is no available standby component that can take over from a failed active component. For this reason, special precautions have to be taken if the period between steps 2 and 7 is protracted. Typically, you utilize a *spare unit* (hardware or software) to do the installation if it requires more time. Spare units are units that are not assigned any active or standby role.

3.4 Cycles in UFC

There may be a case as depicted in Fig.3. The two applications are dependent on data produced by the other one. An example may be that *App2* produces some statistical data based on transaction data produced by *App1*. On the other hand, *App1* is using the statistical data to optimize its own operation.

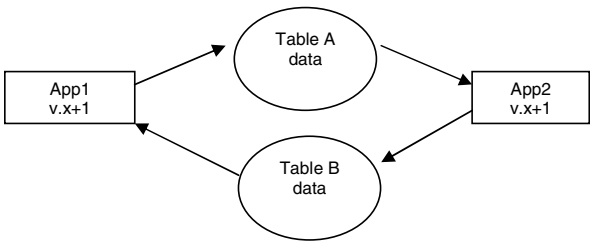

Fig. 3. Example of a cyclic UFC diagram

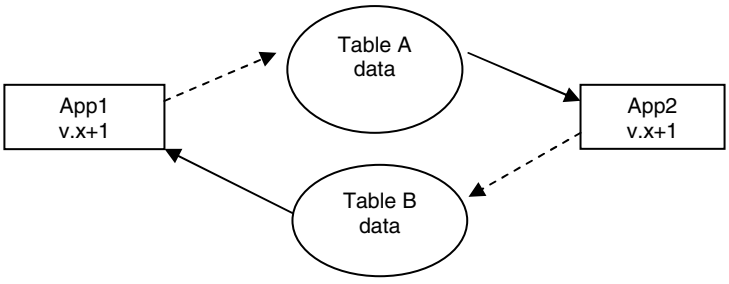

Fig. 4. Introducing weak dependencies (dashed)

If the depicted dependencies are *strong*, i.e. an application cannot operate without the data it is dependent on, we face a problem, because neither application will be able to operate. Therefore, the cycle has to be broken during the implementation of the application upgrade. One way is to implement the upgrade in such a way, that an application may operate, in a limited way, although the new data is not available. In such a case, the upgrade dependency between the application and the data is called a *weak upgrade dependency*.

In Fig. 4, weak dependencies are introduced, allowing to upgrade the two applications in any order.

Requirements. If a UFC cycle is detected, it has to be broken up during the upgrade implementation phase by introducing weak upgrade dependencies. Also, if there are dependencies among components of the same unit, it is preferable to change the dependencies to weak ones, because the actual order of setting the components to the active state may be *a priori* unknown.

Introducing weak dependencies is preferable also otherwise, to ease or remove the ordering requirements in the upgrade execution. There may be, however, some additional cost involved in making components weak-dependable on other components.

3.5 Acquiring and Using UFCs

The information captured in a UFC is mostly based on the incremental changes in the application semantics. If there exists component function dependency information captured in the component metadata similar to the ENT model in [3], the UFC may be extracted automatically by way of differential analysis of the metadata (between the current and the target version). In large diversified systems such metadata is not readily available. Therefore we assume the information pertaining to UFCs have to be acquired from the application developers when they are developing an upgrade. Once UFCs are available they may be used in constructing upgrade scripts to be run on a production system, or even used by an automatic upgrade facility. For this purpose, UFC graphs may be converted to a computer-readable form, e.g. using XML.

3.6 Other Assumptions

In the following sections, when we discuss upgrade scenarios, we make certain assumptions about the quality of upgrades:

- The upgraded software has been tested properly on a test system incorporating all know dependencies.
- The upgrade procedures have been also tested on a test system or on spare units of the production system.
- Because the process of generating UFCs from application semantics is human-centered, and therefore error-prone, one must prepare for the worst and have a backup plan for the situation where the upgrade (despite all proper preparations) is not successful, and the system has be returned to the state, that existed before the upgrade was started. We assume here that system backup images can be and are taken before the start of the upgrade process and that the backup state can be restored if needed.

4 Upgrade Scenarios

4.1 Operating System Upgrade

Operating system upgrade is slightly outside the scope of this paper, as the operating system is, typically, independent of the HA software running in the system. However the capability to perform the service unit switchovers may be utilized in OS upgrades, too. Because installing of a new version of an operating system may be a time-consuming process, spare nodes should be used to perform the installation in the background, without jeopardizing availability of the currently running services. Once the spare is upgraded, the standby node can be brought down and rapidly replaced with the spare, reducing the period of vulnerability of the system.

Similarly to all other software, we assume the compatibility and operation of the new version of the operating system has been tested on a separate test system.

Upgrade Scenario: Operating System

1) Install the operating system on a spare node
2) Install the HA framework, DBMS and applications if necessary.
3) Disconnect current standby node (i.e. the node running standby units) from the active node (resulting in a temporary standalone operation).
4) Transfer the database of the standby node to the upgraded spare node.
5) Assign the spare node the role of new standby node. The old standby node becomes a spare node.
6) The framework initializes the components and the active/standby operation resumes. The active and standby databases become reconnected and resynchronized.
7) Perform a controlled switchover.
8) Repeat steps 1-7 starting with the new spare node and new standby node.

The above scenario should be repeated for all pairs, in a 2N+M redundant system, where M is the number of spare nodes. If there are no spare nodes in a system, the periods of standalone operation will be longer, as the operating system is being upgraded on a standby node.

4.2 HA Framework Upgrade

An HA framework (like SA Forum's AMF) has interconnected instances running on each node. The HA framework upgrades may be dependent on the system model schema updates and new configuration files if they exist (see notes about monotonic schema upgrades in the following subsection). Another difficulty is that all SA-aware (meaning, in the SA Forum parlance, highly available) components are dependent on the framework software because they are typically linked to the framework's libraries. The UFC diagram for framework upgrade is shown in Fig. 5. Because the re-linking the applications make take some considerable amount of time, using of spare nodes is preferable, as in the previous case.

Upgrade Scenario: HA Framework

1) Perform the (monotonic) schema upgrade in the system model database to support the HA framework upgrade (if applicable)
2) Upgrade the HA framework at the spare node.

3) Re-link other SA-aware subsystems and applications with the upgraded framework libraries, at the spare node.
4) Disconnect current standby node (i.e. the node running standby units) from the active node (resulting in a temporary standalone operation).
5) Transfer the database of the standby node to the upgraded spare node (if applicable).
6) Assign the spare node to be a new standby node. The old standby node becomes a spare node.
7) Perform a controlled switchover.
8) Repeat steps 2-7 starting with the new spare node and new standby node.

The above scenario should be repeated for all node pairs, in a 2N+M redundant system, where M is the number of spare nodes.

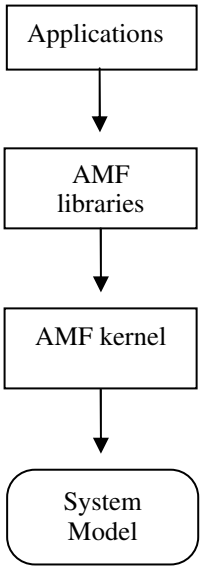

Fig. 5. UFC for HA Framework Upgrade

Requirements. In order for the presented scenario to succeed, the HA framework upgrade has to be engineered in such a way that the instances of the old version and new version of the framework can coexist in the same system. Should this turn out untrue, the rolling upgrade of the HA framework will be impossible, and closing down of the whole system will be required.

4.3 DBMS Upgrade

An HA DBMS must be engineered in such a way that rolling upgrade of the DBMS software is possible. Additionally, the involved dependencies and requirements have to be taken into account. The dependencies related to the DBMS upgrade are shown

in Fig. 6. The weak dependency of applications on upgraded driver libraries (such as ODBC) is explained in the way that the upgraded DBMS should be upward compatible with respect to drivers: the drivers of the old version can be used with the upgraded DBMS. Therefore, drivers may be upgraded at any later time (if a driver upgrade exists). The fact that there is a dependency of applications on new drivers may be explained by possible performance improvements in the drivers.

We assume that the database runs in the active/standby redundancy configuration. Given the assumed short time of performing the upgrade, the scenario does nor employ the spare node.

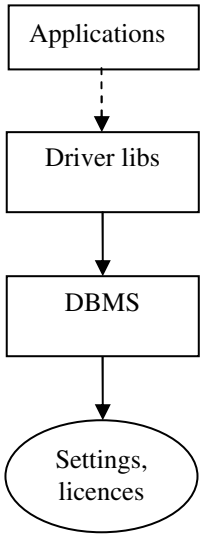

Fig. 6. Dependencies of the DBMS Upgrade

A DBMS upgrade scenario may be very much vendor-specific. The scenario shown below is supported in the Carrier Grade Option of the Solid Database Engine [13].

Upgrade Scenario: HA DBMS

1) Stop the standby DBMS server.
2) Upgrade the DBMS software at the standby node. This involves loading program media, necessary settings and license files into installation directories.
3) Start the upgraded server in the standby mode, with optional conversion mode enabled to convert the database to the format supported by the upgraded DBMS (if applicable). Note: if there are applications that are directly linked to the DBMS, they should be re-linked and restarted, too.
4) Reconnect the servers so they resume the active/standby operation. The necessary database catchup (state resynchronization) is performed automatically.
5) Perform the controlled switchover. The active node runs now the new version.
6) Stop the DBMS server running at the new standby node.
7) Install DBMS at the new standby node.

8) Start the upgraded server at the new standby node, with the optional conversion mode enabled to convert the database to the format supported by the upgraded DBMS (if applicable). Note: if there are applications that are directly linked to the DBMS, they should be re-linked and restarted, too.
9) Reconnect the servers so they resume the active/standby operation, although in the reverse active/standby node configuration. The necessary database catchup is performed automatically.
10) Perform the controlled switchover if the starting active/standby node configuration was the preferable one

Requirements. The crucial characteristics of a DBMS that is needed here is the capability of the new version to maintain the data replication stream with the old version. The minimum requirement is that the upgraded version may take up the standby role while the old version is running as an active. In order to make the upgrade painless for the applications, the new DBMS version must be totally upward compatible with the old one: there should be no change in the old functionality, although new functionality may be added. Also, assuming that there are a set of applications (on other nodes of the system) that should be able to use both older version and the newer version of the database (before and after the switchover), then the newer version of the database server needs to be compatible with the older version of the client API - such as ODBC and JDBC.

4.4 Schema Upgrade

Application upgrades are often dependent on schema upgrades as the new application functionality requires an enhanced data model. Thus, schema upgrades have to be installed before any depending application upgrades. The problem of schema upgrades (or, *schema evolution*) in production systems has been a recognized issue [10]. Typically, the objective of schema evolution is to satisfy the needs of new applications or application updates without jeopardizing the pre-existing applications.

In an HA environment, schema updates have to be performed on a live database, while the applications are running, because bringing the database totally off-line would endanger the overall availability goal. Fortunately, contemporary relational database systems typically support dynamic schema changes. Tables and columns may be added and dropped, referential integrity constraints may be redefined, etc. In an active/standby database pair, the schema changes have to be propagated from the active to the standby database.

Another problem is how to ensure that schema upgrade does not invalidate running applications. To do this, stringent limitations have to be enforced over schema upgrade design and application development. A schema upgrade that is upward compatible with the existing applications (with certain assumptions) is called a *monotonic schema upgrade*.

Definition: Monotonic Schema Upgrade
A schema upgrade is monotonic if and only if:

i. None of the first-class objects[2] is removed or renamed.
ii. None of the existing columns is removed or renamed

[2] First-class objects (in a relational database) are named schema objects created with the SQL CREATE statement, such as tables, views, constraints, triggers, etc.

iii. None of the existing integrity constraints is changed
iv. None of the existing active objects (stored procedures, triggers and events) is redefined

One can see, that a monotonic schema upgrade is, essentially, a schema extension. Objects like tables, views and triggers, table columns and related constraints may be added.

The fact that a schema upgrade is monotonic is not a sufficient guarantee that running applications are not invalidated with the upgrade. The applications themselves have to be built following the *schema-upgrade-safe* rules.

Rules for Schema-Upgrade-Safe Application Development
An application is unaffected by a monotonic schema upgrade if

i. It does not take advantage of any implicit column ordering.
ii. It does not take advantage of table dimensionality (number of columns).
iii. Its error processing (especially of DELETE statements) anticipate possible referential enhancements.

The effect of (i) and (ii) is that statements like SELECT *, and INSERT without explicit columns names, are forbidden. The reason for (iii) is that, as new tables may be associated with existing tables as referencing tables (having foreign keys pointing to existing tables), referential integrity violations may emerge. For example a DELETE statement on an existing table may produce a referential integrity error if there are dependent rows in a referencing table. Normal defensive programming (anticipating errors wherever errors are theoretically possible) will suffice. Additionally, new integrity rules may be added to the new foreign key definitions, like … ON DELETE CASCADE to guarantee that no new referential integrity errors emerge.

Given that the schema upgrade is monotonic and the application are built following the rules for schema-upgrade-safe development, rolling schema upgrades should be possible.

The monotonic schema upgrade should satisfy most needs of the normal application life cycle. Should there be a need for a non-monotonic upgrade involving renaming and changing of the schema semantics, a more careful approach is needed. In such a case, applications have to be scanned for possible change dependencies and reprogrammed accordingly, before the schema upgrade is applied.

The schema upgrade scenario may depend on the HA DBMS implementation used. If an active/standby HA DBMS is capable of propagating the schema changes, as well as data, from the active to the standby database (as does the Solid CarrierGrade Option of the Solid Database Engine), then the upgrade scenario is trivial.

Upgrade Scenario: Schema Upgrade

1) Apply the schema upgrade, dynamically, to the active database. The schema changes are automatically propagated to the standby database.

After creating the new schema elements, such as tables and columns, these are unpopulated (empty). It is often the case that portions of the existing data need to be migrated to the new schema, or that the new schema elements will need some default values or other seed data. Assuming that the applications are developed in a schema-upgrade-safe fashion, e.g. the existing applications can continue using the changed

database schema, the data migration and the new schema population can be applied to the operational active/standby database without causing downtime to service. For example, in the case of Solid Database Engine Carrier Grade Option, data migration tasks would be executed against the active database, and automatically synchronized to the standby, after which the schema upgrade is complete, and both database nodes are ready for use (for the new database client application versions). Note: in the case of large data migration requirements, the data migration itself may have a performance effect and lead to temporary service level degradation. This needs to be taken into account and tested properly when designing the rolling upgrade process.

After the schema upgrade is performed, next come the dependent application upgrades.

4.5 Application Upgrade

Because of possible dependencies, application upgrades should be carefully planned. As some applications may be producers and the other consumers of data, UFC diagrams may useful to capture the dependencies of the type shown in Fig. 2 and Fig.3. The cyclic dependencies have to be discovered early in the upgrade development cycle to allow for reprogramming the applications and introducing weak dependencies. Some of the weak dependencies may be then broken in the UFC graph, allowing for an acyclic graph. An acyclic UFC graph indicates the correct upgrade installation sequence, starting from the outer (leaf) nodes. Note that the ordering of the upgrade is that of partial ordering: any pair of mutually independent upgrades may be installed in any order. If several mutually independent or weakly dependent upgrades are comprised in a single service unit of a HA system, they may be installed in the same installation step. The UFC diagram may be then organized into a set of partially ordered upgrade steps

A single step of application upgrades is performed using the controlled switchover:

Upgrade Scenario: Application Upgrade

i. In a standby service unit of the system, stop the applications awaiting the upgrade.
ii. Install and start the upgraded applications in the standby mode.
iii. Perform a controlled switchover.
iv. Perform steps 1-3 in the new standby unit

Once an installation step is executed, the dependent upgrade steps may be performed. Throughout the time of the upgrade procedure, the applications are continuously available, with the exception of short breaks during switchovers. This way, the goal of providing continuous services is achieved in the presence of system upgrades.

4.6 Other Application System Architectures

In the presentation, we have mostly assumed two-tier (client/server) application architectures. In reality, more complex architectures may be used, including transaction processors, application servers and messaging frameworks like Web Services. In those architectures, the principles of the UFC diagram creation and usage remain the same although new component types may emerge in analysis.

5 Conclusions

Performing rolling upgrades on a continuously operating HA system is a demanding task. It can be successfully performed given proper methods and technologies are used. The prerequisites for a successful rolling upgrade at any level of the system are:

1) Finding out upgrade dependencies and capturing them with, for example, Upgrade Food Chain (UFC) diagrams.
2) Programming the upgrades in a way that allows for existence of weak dependencies and satisfies the rules of schema-upgrade-safe application development.
3) Assuring monotonic schema upgrades.
4) Using an HA DBMS that supports dynamic and uninterruptible schema update.
5) Using an HA DBMS capable of rolling upgrade of the DBMS software.
6) Using a HA framework software capable of doing a rolling upgrade of its own.

There are also several unresolved issues that require further study. Among them are: analysis of the performance impact of rolling upgrades, dealing with ad-hoc environments that do guarantee neither monotonic schema upgrades nor upgrade-safe applications, and satisfying the need to have a possibility to downgrade as well.

References

1. Application Interface Specification, SAI-AIS-A.01.01, April 2003. Service Availability Forum, available at www.saforum.org.
2. Bloom, T., Day, M.: Reconfiguration and module replacement in Argus: theory and practice. Software Engineering Journal, March 1993, pp. 102-108.
3. Brada, P.: Metadata Support for Safe Component Upgrades. COMPSAC 2002: 1017-1021.
4. Brossier, S., Herrmann, F., Shokri, E.: On the Use of the SA Forum Checkpoint and AMF Services. ISAS 2004, May 13-14, 2004 Munich, Germany.
5. Drake, S., Hu, W., McInnis, D.M., Sköld, M., Srivastava, A., Thalmann, L., Tikkanen, M., Torbjørnsen, Ø.,Wolski. A.: Architecture of Highly Available Databases. ISAS 2004, May 13-14, 2004 Munich, Germany.
6. Jokiaho, T., Herrmann, F., Penkler, D., Moser, L.: The Service Availability Forum Application Interface Specification. The RTC Magazine, June 2003, pp. 52-58.
7. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Trans. Software Engineering 18(11), pp. 1293-1306 (November 1990).
8. Van de Laar, F., Chaudron, M.R.V.: A Dynamic Upgrade Mechanism Based on Publish/Subscribe Interaction. COMPSAC 2002, pp. 1034-1037.
9. Moser. L.E., Melliar-Smith, P.M., Tewksbury, L.A.: Online Upgrades Become Standard. COMPSAC 2002, pp. 982-988.
10. Roddick. J.F.: Schema Evolution in Database Systems - An Annotated Bibliography. SIGMOD Record 21(4), pp. 35-40 (1992).
11. Segal M., Frieder O.: On-the-fly Program Modification: Systems for Dynamic Upgrading. IEEE Software, March 1993, pp. 53-65.
12. Solarski, M., Hein Meling, H.: Towards Upgrading Actively Replicated Servers On-the-Fly. COMPSAC 2002, pp. 1038-1046.
13. Solid High Availability User Guide, Version 4.1, Solid Information Technology, February 2004, available at http://www.solidtech.com.
14. Tewksbury, L.A., Moser, L.E., Melliar-Smith, P.M.: Live Upgrades of CORBA Applications Using Object Replication. ICSM 2001, pp. 488.