Dependency-Agnostic Online Upgrades in Distributed Systems

Tudor Dumitraş
tudor@cmu.edu
Carnegie Mellon University
Pittsburgh, PA 15217

Abstract

Traditional approaches for online upgrades of distributed systems rely on dependency tracking to preserve system integrity during and after the upgrade. We propose a mechanism for implementing online upgrades in a dependency-agnostic manner by enforcing the isolation of each version during the upgrade. This allows us to treat the complex IT infrastructure as a black box, with unknown hidden dependencies, and to validate the upgrade by cross-checking the outputs of the two versions.

1. Introduction

An *online upgrade* [1] is a change in the behavior, configuration, code, data or topology of a running application. Online software upgrades are essential for enabling the self-regulating, autonomic management and maintenance of enterprise computer systems.

Online upgrades have to factor in the complex interactions between distributed components and the reliance on specific APIs, networking protocols, queuing paths, configuration settings, etc. For instance, upgrading a component to a version that exposes a modified API (*e.g.*, a modified CORBA object or a WSDL method with different parameters) requires patching all the entities that reference the upgraded component, taking into account the fact that sometimes the old and new APIs may be incompatible. Such dependencies are not always well documented or understood, and are often hard to trace [2]. In general, complete dependency information cannot be detected automatically [2, 3].

Existing approaches for online upgrades simplify the problem by requiring semantic dependencies to be specified by the programmer [1], by constraining the communication to typed message-passing channels [4] or by relying on complete dependency reification [5]. Unfortunately, such constraints render these approaches more difficult to use in practice.

We make the following contributions: in Section 2 we review the open research questions related to online upgrades in distributed systems. In Section 3 we propose a dependency-agnostic approach for online upgrades that does not rely on dependency tracking and does not induce downtime. We achieve this by installing the new version in a "*parallel universe*" – a separate physical or virtual infrastructure that does not communicate directly with the old version. After synchronizing their states, the two versions start executing in parallel and cross-checking their states to validate the upgrade. We also suggest a way for assessing the impact of the upgrades on the running system. We conclude by summarizing our contributions.

2. Research Questions

Dynamic change-management in distributed systems poses a set of unique challenges that are addressed only partially (or not at all) by the current state of the art. While research in online upgrades goes back 30 years [1], these techniques have not gained a wide acceptance in the IT industry due to fundamental problems that currently lack practical solutions.

Most of these problems stem from the complex interactions and dependencies between the distributed components, and the need to quantify and assess the impact of the upgrades, providing performance and dependability guarantees for the future behavior of the system. We focus on the following open questions:

1. **How to avoid dependency tracking?** The dependencies between system components are not always well documented and are very hard to track. An online-upgrading system must be careful not to disable existing applications by breaking unknown dependencies, while updating all the components required by the new version of the application being installed. In general, however, complete dependency information cannot be detected automatically [2, 3]. If this information is specified manually, it may drift from the real state of dependencies; *e.g.* repositories are known to contain metadata inconsistencies leading to version skew [3]. This suggests that, rather than tracking dependencies, it might be preferable to perform upgrades in a dependency-agnostic manner.
2. **What kinds of dependencies are typical in distributed systems?** Empirical studies show that most of the breaking API changes are due to refactorings (modifications of program structure

not intended to change its behavior), which are not always well documented [2]. When upgrading distributed systems, there are additional sources of dependencies, *e.g.* networking protocols, routes middleware. Dependencies cannot be completely discovered through static analysis [1, 5] or runtime monitoring (which is a best-effort approach) [6].

3. **How to handle massive data conversions during the upgrade?** Dynamic upgrades must preserve the correctness of the system, which often requires the transfer of state composed of persistent and even transient data. Many applications require massive amounts of data to be converted to new schemas, which happens over a long period of time during which clients may request transactions involving the same data being converted [7].
4. **How to perform upgrades across mutually-distrustful administrative domains?** Sometimes upgrades span multiple administrative domains, which may have different security policies. Moreover, an upgrade propagating from another domain is hard to distinguish from an intrusion, since the upgrade may modify the same critical files that viruses and worms are likely to be interested in.
5. **How to upgrade autonomic and self-managing distributed systems?** Modern distributed systems are often built using independent off-the-shelf components under the control of component-specific configuration-managers that use extensive, and sometimes proprietary, domain knowledge (*e.g.* workload characteristics, resource-utilization models). In such environments, it is unfeasible to build a fully-centralized upgrade planner that doesn't have access to this domain knowledge [8].
6. **How to assess the impact of long-running upgrades on a system with varying workloads?** Dynamic upgrading systems must assess the impact of the changes on the running services and determine the most opportune moment to apply the upgrade to avoid significant penalties due to degraded performance and dependability, while improving the value of the infrastructure according to some well-defined metrics [8].
7. **How to provide quantifiable dependability and performance guarantees?** The upgrading process must be reliable and tolerate faults without the loss of data or functionality, including in the case when upgrades are interrupted by faults.

3. Description of Approach

We propose a dependency-agnostic online-upgrade approach that does not rely on dependency tracking and does not induce downtime. Instead of an in-place upgrade, we aim to isolate the new version from the old. The new version, with a potentially different topology, is the result of a fresh installation and does not communicate directly with the old version. The persistent data is injected into the new version while the old version continues to service requests. We avoid data staleness by detecting and re-transferring data items that have changed during the upgrade. We leverage the fact that most enterprise systems have a few well-defined ingress and egress points (*e.g.* the URL where clients send their requests and the database where the persistent data is stored). By intercepting the request flows at these points we can collect sufficient information for maintaining the data consistency, while treating the rest of the infrastructure as a black-box. While the two versions keep running in parallel, administrators can cross-validate the outputs and roll back a failed upgrade.

This approach targets major upgrades in medium-to-large enterprise infrastructures. These systems may have intricate dependencies such as: component-level dependencies (*e.g.* version of database software required), configuration-file dependencies (may specify shared libraries to load or changes in behavior), protocol dependencies (*e.g.* HTTP, SQL, proprietary protocols), semantic dependencies (*e.g.* HTTP requests triggering SQL queries), performance dependencies [7].

The protocol has four stages:

Bootstrapping. The biggest problem for the bootstrapping stage is capturing in-progress updates. This problem is aggravated by the presence of caches at various tiers in the infrastructure, which may delay the insertion of the update into the database. To guarantee that no updates are overlooked, we may flush all the caches from the old system, or, as a last resort, restart the entire infrastructure, with the same effect.

Data Transfer. During this phase, we convert the data from the old version into the schema used by the new one. The mapping between the two schemas must be specified in advance, before starting the upgrade. Based on this mapping, we attempt to find the closest equivalent of a data item in the new database. This mapping is not exact because there may be semantic differences between the two data formats. Moreover, some data items that have identical semantics cannot be transferred (*e.g.* hashed passwords). We maintain a list of items to track the entries that have already been transferred. When the egress interceptor detects that a certain data item is updated, its corresponding entry in the page list is invalidated and the item is (re)scheduled for transfer. The data transfer will eventually terminate if the transfer rate exceeds the rate at which converted data is invalidated.

Parallel Execution. After the database transfer is complete, the two universes may enter the parallel-execution stage. The middleware freezes the state of the old version by blocking update requests (these are queued and applied later). When all the in-progress updates have been committed to the database and transferred to the new system, the persistent states of the two universes are in sync and the two universes can start executing in parallel. Requests intercepted at the ingress point are injected into both versions of the system, after another conversion step. In this stage, we can compare the two outputs in order to validate the upgrade's integrity. The clients receive only the output from the master universe (initially, the old system).

Switchover. The switchover changes the master universe from the old to the new version. We discard volatile state, such as user sessions, and users will be required to log in again. After the switch, the two universes continue to execute in parallel, which allows system administrators to roll back the upgrade, if needed, without loss of data. In fact, as long as the two systems continue to execute in parallel, we can switch back and forth between the versions. However, the two states will not be perfectly synchronized anymore because of behavioral differences between the two systems. This state divergence is acceptable (and even desirable) because the new behavior could be the very reason of the upgrade.

Dependency-agnostic upgrades are not surgical procedures and are likely unsuitable for regular maintenance activities such as applying security patches. This approach is most appropriate for large-scale, distributed upgrades because it eliminates downtime and it reduces the administrative burden of identifying dependencies. The data transfer and conversion of the entire database from the old system is a long-running process [7] that might disrupt the normal operation of the system during high load. We therefore need a way to assess the impact of the upgrade on the performance and dependability of the system, both during and after the upgrade. We can use these predictions for reducing the data transfer rate during the busy hours and for throttling back the two versions in the parallel execution stage to ensure that they can proceed at similar speeds.

We have proposed an impact-sensitive framework for dynamic change management [8], which allows us to schedule upgrades based on their impact on the IT environment. This framework makes minimal assumptions about the kind of probes and hooks available to the online upgrader by separating and distributing the upgrade-scheduling and the impact-analysis. In our framework, these tasks are performed by different components, which may come from different providers.

For instance, after entering the parallel execution stage, the states of the two systems (which provide similar, but not identical functionality) will start diverging. In order to validate their outputs and to determine if the upgrade was successful, we must use analytical models asserting the correctness and desirability of the observed behaviors. These models are often based on proprietary domain-knowledge that the vendors may not be willing to expose. Instead, the vendors provide "advisors" for validation checking, performance prediction, problem determination, etc. that encapsulate the required domain knowledge. In our framework, a centralized component queries all the advisors and computes the optimal schedule for the upgrade (which may include a roll back if necessary).

4. Conclusions

We propose a dependency-agnostic approach for performing major upgrades in complex distributed systems. This technique enforces the isolation between versions by installing the new system on a parallel universe and transferring data in the background. By running the systems in parallel and comparing their outputs, we can validate the upgrade and roll back if needed. This approach allows us to treat the entire IT infrastructure as a black box with respect to hidden dependencies between distributed components, while assessing the impact of the upgrade on system performance and dependability.

References

[1] M. Segal and O. Frieder, "On-the-fly program modification: Systems for dynamic updating," *IEEE Software*, vol. 10, pp. 53-65, Mar 1993.

[2] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, pp. 83 - 107, Mar/Apr 2006.

[3] J. Hart and J. D'Amelia, "An analysis of RPM validation drift," in *Large Installation System Administration*, Philadelphia, PA, Nov 2002, pp. 155-166.

[4] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1293-1306, 1990.

[5] F. Kon and R. H. Campbell, "Dependence Management in Component-Based Distributed Systems," *IEEE Concurrency*, vol. 8, pp. 26-36, 2000.

[6] J. Dunagan et al., "Towards a Self-Managing Software Patching Process Using Black-Box Persistent-State Manifests," in *International Conference on Autonomic Computing*, 2004, pp. 106-113.

[7] T. Dumitraş et al., "No More HotDependencies! A Case for Dependency-Agnostic Online Upgrades in Distributed Systems," in *Workshop on Hot Topics in System Dependability*, Edinburgh, Scotland, Jun 2007.

[8] T. Dumitraş et al., "Ecotopia: An Ecological Framework for Change Management in Distributed Systems," in *Architecting Dependable Systems IV*, *LNCS*, C. Gacek and *et al.*, Eds.: Springer Verlag, 2007.