Cloud Software Upgrades: Challenges and Opportunities

Iulian Neamtiu
Department of Computer Science and Engineering
University of California, Riverside
Email: neamtiu@cs.ucr.edu

Tudor Dumitraș
Symantec Research Labs
Symantec Corporation
Email: tudor_dumitras@symantec.com

Abstract

The fast evolution pace for cloud computing software is on a collision course with our growing reliance on cloud computing. On one hand, cloud software must have the agility to evolve rapidly, in order to remain competitive; on the other hand, more and more critical services become dependent on the cloud and demand high availability through firm Service Level Agreements (SLAs) for cloud infrastructures. This race between the needs to increase both the cloud upgrade frequency and the service availability is unsustainable. In this paper we highlight challenges and opportunities for upgrades in the cloud. We survey the release histories of several cloud applications to analyze their evolution pace, and we discuss the shortcomings with current cloud upgrade mechanisms. We outline several solutions for sustaining this evolution while improving availability, by focusing on the novel characteristics of cloud computing. By discussing several promising directions for realizing this vision, we propose a research agenda for the future of software upgrades in the cloud.

1. Introduction

Cloud computing is increasingly popular, owing to its computing-as-a-utility nature, which requires no up-front equipment cost, relieves administration burden and provides the perception of unlimited resources. Business applications have an incentive for moving into the cloud, to reduce the operational and maintenance costs. The cloud can also be used a platform for scientific discovery, by allowing researchers to develop and run massively scalable scientific applications with little to no up-front investment.

Many critical systems depend on cloud services. The U.S. military is equipping soldiers with access to military-run cloud computing for "critical surveillance and decision-making information" [1]. Educational institutions are using Facebook and Twitter notifications to broadcast emergency alerts. Healthcare records and patient monitoring data are moving into the cloud, e.g., by using Microsoft's Amalga, a cloud-based service for storing and accessing patient data [2], in use at hospitals in the United States and other countries. A recent cloud outage has emphasized our reliance on cloud computing: on the morning of April 21, Amazon's Northern Virginia data center started having connectivity and failure issues with its Elastic Block Store (EBS), and Elastic Compute Cloud (EC2) instances; while most of the problems were resolved within 24 hours, the complete recovery took until April 25. This outage has affected online sites such as Foursquare, Reddit and Quora, which depend on Amazon's cloud [3], [4]. The root cause of the outage was traced to an operator error during a routine upgrade meant to increase capacity [5].

These critical services are driving the demand for improvements in cloud availability. *Availability* is the fraction of time that a system is ready to provide correct service and does not experience planned or unplanned downtime. Availability can be subject to contractual obligations; in particular, Service Level Agreements (SLA) for business applications require that cloud service and applications be highly available. For example, the Google Apps for Business SLA stipulates that applications be available 99.9% of the time, otherwise Google is liable to compensate users for unavailability [6]. At the same time, cloud software tends to evolve at a rapid pace, and cloud software upgrades must be deployed frequently to add new features or to fix security vulnerabilities. Therefore cloud service providers need to apply software upgrades while sustaining service, a procedure called *online software upgrade*.

Similar concerns for service availability provided the motivating force for the initial development of online upgrade techniques, in the telecommunications industry. For example, AT&T's 5ESS telephone switch was one of the first computer systems capable of updating the running program on-the-fly, without service interruptions [7]. The inclusion of these capabilities was driven by the stringent availability requirements of the public switched telephone network—five nines (99.999%) of availability, i.e., less than 6 minutes of downtime per year—because 5ESS was developed as a part of the network infrastructure, rather than as a stand-alone product. By comparison, Gartner estimates that current enterprise applications with "very good availability" allow 200 hours of planned downtime and 61 hours of unplanned downtime per year (i.e., less than *two nines* of availability) [8].

Recent studies and a large body of anecdotal evidence suggest that software upgrades are one of the leading causes of such downtime. A 2007 survey of 50 system administrators from multiple countries (82% of whom had more than five years of experience) concluded that, on average, 8.6% of upgrades fail, inducing unplanned downtime, with some administrators reporting failure rates of up to 50% [9]. Most of these failures can be traced back to accidentally breaking hidden system dependencies during the upgrade [10]; moreover, software upgrades often require planned downtime as well [11].

Industry best practices for software upgrades recommend avoiding downtime through *rolling upgrades*, which upgrade and then reboot each host in a wave rolling through the data center [10], [12]. For example, Azure—Microsoft's cloud service—employs rolling upgrades by partitioning the cloud into "update domains" and performing updates one domain at a time (which requires shutting down and restarting the virtual machines in each domain) [13]. Such rolling upgrades avoid service interruptions and impose very little capacity loss, but they are feasible only for implementing changes that maintain backward compatibility. This limitation stems from the fact that the system's clients can interact with either the old version or the new version of the software during a rolling upgrade, which requires the two versions to interact with each other in a compatible manner. Current commercial tools that facilitate rolling upgrades provide no way for determining if the interactions between mixed versions are safe and leave these concerns to the application developers [14], [15]. In general, the behavior of a system with mixed versions is not guaranteed to conform to the specification of either version and is hard to test and validate in advance [16]. Moreover, the failures that arise from such component-wise upgrades are not well understood, as prior research on online upgrades focused on evaluating performance and overhead, rather than on the upgrade dependability [10].

2. Cloud Software Evolution

Cloud computing has introduced new paradigms that are not well integrated into the existing approaches for software maintenance and evolution. For example, the ability to adjust the compute and storage resources elastically, which is one of the most distinctive features provided by cloud infrastructures, is not fully utilized by the current upgrade mechanisms.

Furthermore, cloud-based distributed systems ship client-side code to users whenever they connect to the service (e.g., through AJAX-based Web programming). This reduces the concerns for client-side software evolution [17], because the system is designed from the start to be able to load new code whenever needed. It also reduces the concern for disseminating updates on an Internet scale [18], because the application logic is implemented on the server side and executes inside the service provider's data center. This focus on the server side is compelling for service providers because of the need to support resource-constrained mobile devices and because adequately testing thick clients would require longer release cycles [19].

Conversely, frequent patches and updates are required on the server side, to introduce new features and to fix bugs, for allowing service providers and application developers to remain competitive. For example, Facebook employs tight release cycles with application updates being "pushed" (deployed) *more than once a week* [20]. The release interval decreases drastically (e.g., from days to hours) when new updates must be developed and applied to fix security vulnerabilities.

The cloud provides three levels of abstraction: infrastructure-as-a-service, for running custom software stacks; platform-as-a-service, for developing and deploying web-based applications; software-as-a-service, for complete applications such as email and document management:

1) **Infrastructure as-a-Service (IaaS)** provides access to virtualized resources (e.g., CPUs, block storage, key-value stores, SQL databases) and gives customers complete flexibility to run their own software stacks on top of these resources. Examples of IaaS include the Amazon Web Services—e.g., Elastic Compute Cloud (EC2), Simple Storage Service (S3), Elastic Block Storage (EBS)—and the Windows Azure Fabric.
2) **Platform as-a-service (PaaS)** provides a middleware service or runtime system on top of IaaS. For example, Windows Azure AppFabric

Table 1. Evolution of cloud applications and of their components.

Application/ component	Time frame	Releases	Average release interval (days)
Sumo Paint	6/2008–10/2010	63	13
Facebook (platform)	5/2008–12/2010	>138	<7
Google Docs (List API)	02/2009–11/2010	>12	<51
MediaWiki (Wikipedia)	04/2003–11/2009	116	21
Tiki Wiki	11/2002–11/2009	43	60
Slash	11/2001–11/2009	135	22
Joomla	11/2007–11/2009	25	30
NCBI BLAST	01/1999–01/2011	33	136
Velvet	11/2007–11/2010	71	16
Memcached	07/2003–04/2010	37	68
SQLite	08/2000–02/2009	172	19

or Google App Engine provide APIs for building customized applications, including cached datastores, URL fetch, multi-tenant support, component cloning and automated migration for reducing latency.

3) **Software as-a-service (SaaS)** provides complete applications that run in the cloud and that are accessed through thin clients (e.g., in a browser). For example, Google Apps for Business provides organizations with cloud services for email, calendar, document storage and collaboration.

To illustrate the evolution of cloud applications, we have collected data on how software components used at various levels (IaaS, PaaS, SaaS) change over time. Table 1 summarizes this data. The programs we analyzed span existing cloud applications (Sumo Paint, Facebook, Google Docs), software components commonly used in cloud applications (MediaWiki, Tiki Wiki, Slash, and Joomla), high-performance computing applications that are transitioning to the cloud (NCBI BLAST, Velvet), and components of the cloud infrastructure (Memcached and SQLite). For each program, the table presents the analyzed time frame, the number of official releases, and the average number of days between successive releases. As we can see, new official versions are released every 7 to 135 days; note that the number of releases is a lower bound, as many patches might be pushed in-between official releases.

We now proceed to briefly describing each application. Sumo Paint is a full-featured cloud-based photo editing/painting application [21]; as we can see in Table 1, Sumo Paint updates are released on average every 13 days. According to their developer blog [20] Facebook "pushes" new versions of their platform software at least once a week. Google Docs is the popular Web application from Google that allows documents to be stored in the cloud and edited collaboratively. Google Docs (List API) shows how frequently a subset of the Google Docs API changes. MediaWiki, Tiki Wiki, Slash and Joomla are wikis and content management systems, used for online collaboration; we synthesized their evolution data from Mandalapa's work on schema evolution [22]. NCBI BLAST [23] is a popular genomic sequence alignment program that has been shown to scale very well in the cloud [24]. Velvet [25] is a popular, resource-intensive genomic sequence assembler that has recently been ported to OpenMP and that we envision will transition into the cloud. Memcached is a high-performance, distributed-memory cache, used on high-traffic sites such as YouTube, Facebook, and Wikipedia to relieve database hot-spots by storing and delivering pre-rendered Web content. SQLite is a server-less, zero-configuration SQL engine [26] that is often used by cloud clients to cache data locally. It is important to point out that client-perceived availability depends on the availability of the entire software stack, hence upgrade-induced downtime to *either* infrastructure, platform, or service will lead to client-perceived unavailability.

3. Challenges and Opportunities for Cloud Upgrades

The existing techniques for upgrading software online are not always appropriate in the cloud setting, because they do not properly address the new interaction patterns among cloud infrastructure components and they do not take full advantage of the new mechanisms provided by the cloud. We begin by looking into the reasons why upgrades cause unavailability (Section 3.1). We analyze the shortcomings of rolling upgrades—a technique commonly used for avoiding upgrade-induced downtime in distributed enterprise systems—such as their inability to support major changes to the structure of persistent data (Section 3.1) or upgrades to communication protocols (Section 3.3). We also illustrate the inconsistencies introduced by rolling upgrades in a cloud computing environment that spans multiple administrative domains—a scenario that can occur at many layers in the new cloud architectures (Section 3.2). The fast evolution pace of cloud software poses development and update-safety analysis challenges; we show how a new approach, explicit support for evolution in programming languages and tools, could alleviate this development and analysis burden (Section 3.4). We present the opportunities for utilizing cloud-specific mechanisms, such as the elastic allocation of resources, for providing an alternative to rolling upgrades and for testing upgraded systems in an environment that faithfully mirrors the deployment setting (Section 3.5). Finally, we emphasize the need for establishing a benchmark for the dependability of software upgrade mechanisms, based on field-gathered data on upgrade failures (Section 3.6).

3.1. Why Do Upgrades Cause Unavailability?

The rolling upgrade approach was developed for enterprise systems with fixed resources [12] and is not well suited for cloud computing, where the resources available to an application can be scaled up and down on demand. For example, upgrading an infrastructure, platform or application *in place*, through a rolling upgrade, leads to a temporary performance degradation. This degradation is due to (i) the loss of capacity when nodes are restarted, and (ii) the overhead of the upgrade operations, e.g., converting the persistent data to a new schema. Such performance degradations could lead to violations of the SLA provisions for latency and throughput. In practice, this is often considered an acceptable trade-off because rolling upgrades are perceived to reduce the risks of upgrading, as failures are localized and might not affect the entire distributed system [15], [27].

However, rolling upgrades cannot implement certain types of changes that require modifications to persistent data structures on disk. As an example, we study the complex schema changes that have been necessary during Wikipedia's upgrade history [11]. Wikipedia stores its persistent data on several MySQL database servers, configured for master/slave replication.[1] Wikipedia tries to deploy these changes through rolling upgrades in the back end. Such a rolling upgrade removes slave nodes one-by-one from the replication group, applies the schema changes, and then restarts the replication. The rolling upgrade swaps database masters before completing the schema upgrade, to avoid re-applying the changes through the replication mechanism.

Online upgrades are not always feasible in the back end; for example, a 2004 upgrade from MediaWiki

1. The master database receives the write queries and propagates the updates to the slaves, which handle the read-only queries that constitute the bulk of Wikipedia's workload. In April 2009, Wikipedia had 23 MySQL servers.

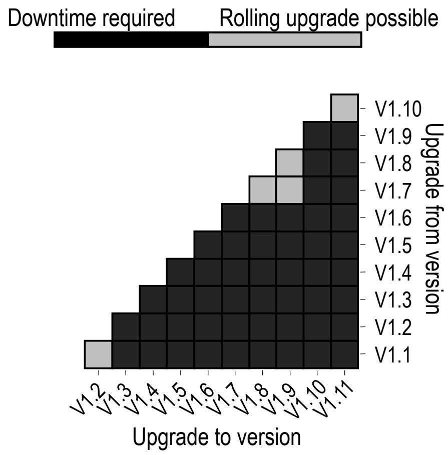

Figure 1. Planned downtime imposed by MediaWiki upgrades.

version 1.4 to version 1.5 required 22 hours of planned downtime [28]. To support rolling upgrades, the database replication mechanism must allow source and target tables that do not have identical schemas, and the schema changes must preserve backward compatibility. For example, in MySQL, a table on the master node can have more or fewer columns than the slave's copy; however, dropping or renaming columns prevents a rolling upgrade because the old application would be unable to query the new schema during the upgrade. Adding a new column to an existing table, which is the most common schema change in Wikipedia [29], is usually compatible with rolling upgrades, except when creating columns with values incremented automatically (this might not produce the same ordering of rows on the master and the slave). As illustrated in Figure 1, out of the 55 possible upgrades among MediaWiki versions 1.1–1.11 only 5 can be executed online, through a rolling upgrade [11]. Schema evolution in other open-source server systems shows similar trends; for example 50% of schema changes in the Monotone version control system are table and column deletions [30], [31].

Upgrading commercial systems can require even more complex schema changes. Examples of changes from Oracle's enterprise systems include accessing multiple rows in the source table (through a self-join), creating a primary key from a column that allows repeated values, and initializing new columns with aggregate values, which are difficult to maintain incrementally, in response to updates from the live workload (e.g., computing `MAX(`*column*`)` when values from *column* may be deleted by the live workload) [27]. These challenges are not limited to databases, but they also affect upgrades that modify other persistent data-structures, such as the metadata used by distributed file systems. For example, new versions of the GPFS parallel file system [32] are usually deployed without downtime, through rolling upgrades. However, when GPFS's inode structure was updated to 64-bit disk sector numbers (from 32-bit), the upgrade required unmounting the file system for changing the metadata on disk [33].

Such levels of planned downtime are likely unacceptable for cloud services with SLAs that specify well-defined penalties for violating the availability target. At Facebook, for example, changes to database schemas are usually limited to adding columns and tables, and schema inconsistencies between the application and the database do not constitute a significant challenge [34]. This is the result of Facebook's highly-connected user base. Because the friendship connections evolve continuously and do not produce stable clusters, the Facebook system scales better through horizontal partitioning (e.g., by splitting users across several databases) than vertical partitioning (e.g., by splitting the names and addresses of users in different database tables). This allows Facebook to avoid the planned downtime required to implement major schema changes. However, for other cloud computing applications, avoiding complex schema changes, which might impose an unacceptable downtime, leads to the preservation of database schemas that provide sub-optimal performance and that cannot support new, user-requested features [11].

Furthermore, we previously showed that, contrary to conventional wisdom, rolling upgrades are prone to failure because they place the system in a state with mixed versions, which increases the risk of breaking hidden dependencies during the upgrade [10]. Such broken dependencies represent the leading cause of unplanned downtime resulting from software upgrades.

3.2. The Many Sources of Inconsistency in Cloud Upgrades

In addition to planned and unplanned downtime, rolling upgrades expose the system to a type of race condition that is specific to mixed-version states. Such mixed-version races occur in multi-tiered systems where a consistent upgrade schedule cannot be enforced for all the tiers. Asynchronous message exchanges among tiers potentially lead to a situation where an invocation from the new version is processed by the old version on a different tier of the application.

For example, in Web applications that use the AJAX programming model, the client-side code, running in the user's browser, periodically issues asynchronous callbacks into the server (using the `XMLHttpRequest` mechanism) to request additional data and to refresh the page. During a rolling upgrade on the server side, the system enters a state where both server versions (old and new) co-exist in the cloud front-end. In this state, user interaction might cause the browser to load the new version of the client-side code, and callbacks issued by this code can arrive at a front-end server that has not yet been upgraded and that continues to run the old version of the server-side code. If one such callback was added or modified during the upgrade, there are two possible outcomes: (i) the old version of the server-side code does not know how to handle the request and returns an error to the user; or (ii) the sever-side code does not detect a malformed request (e.g., because the upgrade changed the semantics of the callback, but not the API) and causes a silent inconsistency. Such *mixed-version races* occur frequently during cloud system upgrades, and they can have a severe impact [35].

Mixed-version races can be prevented by completing the server-side upgrade before upgrading the clients in a multi-tiered system [36], to prevent the new version from calling into the old version. These approaches are infeasible in distributed systems that communicate across *multiple administrative domains*, with no central upgrade coordination. This is a common scenario for AJAX applications, which rely on client-side code, and for enterprise systems that lease cloud computing resources (e.g., in an IaaS setting) or that incorporate components executing in the cloud (e.g., in a PaaS setting or when utilizing components from more than one cloud provider to build a mashup application).

The effects of mixed-version races are an example of the inconsistencies that can occur when upgrading a cloud computing system. The safety of cloud upgrades is threatened by many similar inconsistencies, introduced while upgrading nodes, tiers, or data centers that span multiple administrative domains. Figure 2 illustrates the four levels where these inconsistencies may occur: when applications communicate across nodes within the same tier, across datacenters, across tiers, and across clouds.

Maintaining cross-node consistency. Section 3.1 shows that 90% of historical MediaWiki upgrades require planned downtime, because the database changes introduced would lead to inconsistencies during an online upgrade. A similar scenario occurs when upgrading running memcached instances within a datacenter, as all instances assume the same internal data representation and communication protocol. We stud-

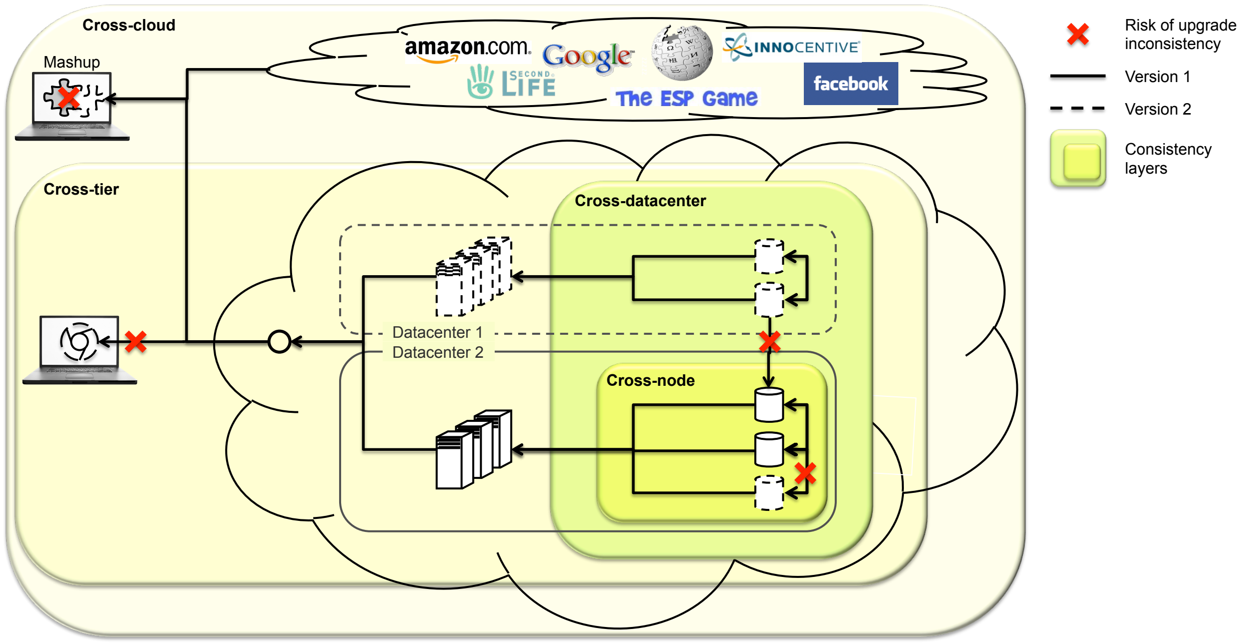

Figure 2. Upgrade inconsistencies can occur in four layers of cloud computing systems.

ied the evolution of memcached during the 10 months between May 2007 and March 2008, when 14 function signatures and 6 data types have changed [37]. These changes may violate the assumptions of cross-node communication during the upgrade. Upgrading one node to the next memcached release—that implements a different protocol and uses a different data format—can lead to failures and inconsistencies. We presented a solution to this problem in our prior work on safe dynamic updates to cloud applications [38]. In this approach, programmers mark memcached blocks of code that must execute in a *version-consistent* manner (i.e., the entire block appears to execute at the old or new version, but not both); an update agent will then ensure that updates preserve version consistency, without requiring synchronization across all nodes.

Maintaining cross-datacenter consistency. Ensuring consistency is even more challenging when a tier spans multiple data centers. For example, service providers allow cloud applications to store data in multiple data centers that are geographically separate, in order to place the data closer to the users and to provide fault isolation. Whenever the data format at one data center is upgraded, a format incompatibility might arise between the two data centers if the data centers are not upgraded simultaneously.

Maintaining cross-tier consistency. The recorded occurrences of mixed-version races [35] demonstrate that online upgrades can lead to cross-tier inconsistency scenarios. In general, multi-tier applications have precise expectations about the types and sequences of data objects that cross tiers; for example, these objects must have common semantics when implemented in JavaScript on the client, PHP on the compute tier and SQL on the storage tier. The fast evolution pace that characterizes cloud computing applications can lead to divergences in semantics; this problem, known as *impedance mismatch*, poses a development-time challenge. In addition, certain inconsistencies manifest only at deployment time (e.g., as in the case of mixed-version races) and are difficult to detect during testing. Solutions that have been proposed for maintaining cross-tier consistency include using a single language for all three tiers, to avoid impedance mismatch during development [39], and forcing the upgrade to begin at the storage tier, continue with the compute tier, and finally the client, to avoid deployment-time inconsistencies [36]. Another solution is to make mixed-version interaction explicit in the implementation, which will make application upgrades safe by construction. We develop this idea further in Section 3.4.

Maintaining cross-cloud consistency. When applications communicate with multiple clouds, we cannot assume that all the cloud-based components will be updated in unison. Consider, for example, a mashup application that aggregates components from different providers. A change in the format of one mashup source (as a result of an upgrade at one provider) can lead to incompatibility with other mashup sources because their components do not yet reflect the changes.

These inconsistencies arise from the mixed-version states created during rolling upgrades and from the attempt to replace existing components in place. Cloud computing provides the opportunity to avoid these shortcomings by increasing the amount of compute and storage resources available during the upgrade. Such additional resources can provide a *parallel universe* for installing the new version and for performing all the operations required by the upgrade (including the computationally-intensive data and schema conversions) [10]. This solution supports complex changes to data-intensive systems (e.g., the schema changes that have caused planned downtime for Wikipedia [11]), by decoupling data conversion from normal system operation, and avoids mixed-version races, by upgrading all the application tiers atomically. By trading the resource overhead for improvements in upgrade dependability, this approach harnesses some of the unique features of clouds to address challenges that are specific to cloud computing.

3.3. The Next Upgrade of Communication Protocols

Upgrading communication protocols online requires changing all the endpoints without interrupting the existing connections. This is especially challenging for applications with long-running transactions that cannot tolerate even brief connection interruptions. These applications include familiar examples, such as voice-over-IP, which share many of the characteristics of the telecommunication systems that motivated the initial development of online upgrade mechanisms. However, the adoption of cloud computing enables a new class of applications with long-running transactions. For example, this paper was written with the aid of Google Docs, an AJAX-based application that allowed the authors to edit the content and to review each other's changes in real time, despite not being physically co-located. Such collaborative applications require a regular exchange of messages between the server and the endpoints. The traditional approach for upgrading such applications is to upgrade one half of the server-side nodes, to direct new connection requests to the new version while waiting for all the connections to the old version to drain, and then to upgrade the second half [7]. Generic techniques for modifying the communication protocol on-the-fly and for transferring the connection state to the new version have not yet been developed; such techniques typically rely on specific domain knowledge about the protocol's semantics [40], [41].

An alternative approach is to allow servers to submit the protocol stubs to the clients and to render the clients protocol-agnostic—as pioneered by the Jini middleware [42]. In the past, this has allowed Orbitz, an airline ticketing system and one of the early adopters of Jini technology, to perform seven major upgrades without failure and without downtime [43]. This approach can be revisited in the context of cloud computing. Cloud-based distributed systems are able to send the appropriate client-side code to the users whenever they connect to the service (e.g., through AJAX-style programming, but other forms of code migration might emerge in the future). For example, the Facebook Connect and Google Friend Connect protocols, which allow a user to share his or her social-network identity with third party sites, implement the entire cross-domain communication inside the user's browser through an HTML5 technique commonly used in mashup applications (`postMessage`, which allows exchanging primitive strings among iframes with different origins) [44].

3.4. Explicit Support for Evolution: An Idea Whose Time Has Come

Making evolution a first-class language feature is a way forward towards sustainable, correct cloud software evolution. Forcing programmers explicitly to reason about the presence of multiple versions (e.g., as created by rolling upgrades) instead of fixing version inconsistencies introduced by upgrades, is a good first step toward supporting the fast evolution pace of cloud software. In particular, explicit versioning of code and data and first-class expression of mixed-version interaction are missing in current programming languages.

Explicit versioning of code and data. Types, functions, database schemas should have attached version numbers, and any evolutionary change should clearly point out of the ways in which the new version differs from the old version. For example, a database change that adds attributes is backward-compatible, but, without inspecting the table creation code for both the old and the new version, programmers may be unaware of this change and may fail to initialize the new attributes correctly. If this attribute addition is made explicit in the code that accesses the schema, it allows programmers to reason about upgrade safety. Note that relying on tools [29], [45] that reconstruct evolutionary changes (i.e., differences between versions) addresses this problem only partially. Such tools are not always able to detect non-trivial changes, e.g., renaming of fields/methods/functions or complicated schema evolution operations.

First-class expression of mixed-version interaction. When cloud compute code version N tries to access cloud storage data version M, the repercussions of this mixed-version interaction should be clearly documented as either forbidden, permitted with restrictions (e.g., only access a subset of storage attributes), or permitted without any restrictions. This would facilitate reasoning about the safety of rolling upgrades, e.g., whether mixed-version races may occur, or what happens when a compute node running at version N tries to communicate with another compute node running at version N$\pm$1.

Current trends in cloud software development emphasize the need for such features. For example, Facebook's Gatekeeper tool for source code management and development uses explicit in-line versioning for the source code, rather than relying on version control systems to manage different branches [34]. While the goal of Gatekeeper is to decouple the deployment of new features from their activation, a by-product is that it facilitates the programmers' reasoning about the effects of two different coexisting code versions running in the cloud. For example, while a rolling upgrade is in progress, a node will run version N while some other node runs version N+1; if both access the database, having code versions N and N+1 in the same source file will allow the programmer to reason about breaking changes. With existing programming language constructs, this is achieved by writing tangled `if`-ladders that render the code difficult to understand.

Recent work has explored better constructs for expressing evolution. For example, UpgradeJ [46] makes evolution a Java language feature: class names have explicit version annotations and two new language constructs, `revises` and `evolves` are used to either revise the implementation of existing methods, or to evolve existing classes with new methods and fields. However, these constructs are not powerful enough to handle the real-world software evolution: UpgradeJ's mechanisms can support up to 65% of historical changes to Java programs (considering the 11 applications in the Qualitas Corpus, a collection of open-source Java projects that span multiple releases) [47]. However, these numbers represent a worst-case assumption, because these projects were developed without explicit language support for evolution.

Dynamic software updating (DSU) can also be used for online upgrades in the cloud, while requiring developers to reason about the safety of upgrades and explicitly handle changes between releases [38]. With DSU, programmers develop software normally, by only focusing on one version at a time. The initial version of an application is compiled specially into code that accepts dynamic updates by loading a dynamic patch. When a new version is ready, the differences between the old and new versions are encapsulated in a dynamic patch. An upgrade consists of loading the dynamic patch into the old (running) program. In this approach, a special compiler transforms existing programs into programs that accept runtime updates, and automatically generates most of the dynamic patch. Another DSU approach, which does not require a special compiler, is to transfer live program state between versions using checkpointing [48]. DSU approaches compel the developers to reason about evolution, because they need (i) to specify program points where updates are permitted to ensure update safety, and (ii) to write code that transforms program state from the representation used by the old version into a format expected by the new version.

3.5. The Art of Testing Cloud Systems

Software upgrades often fail and induce unplanned downtime due to differences between the environments in which a system is tested and deployed [9], [49]. Moreover, the tight release cycles of cloud computing systems provide limited opportunities for testing the new version and the intermediate steps of the upgrade. As a result, the changes implemented through online upgrades interact with the workload in ways that are unpredictable, or that cannot be tested exhaustively at design-time (e.g., mixed-version races). Reasoning about such runtime-emerging behavior is difficult because previously-established system invariants do not hold, changes are implemented by both human and software agents, hidden dependencies in the environment can induce upgrade failures, and externally-imposed deadlines might affect the outcome.

At the same time, the cloud provides new opportunities for testing. For example, the snapshotting features provided by cloud storage and the record-and-replay features of virtual infrastructures empower the software engineers to reproduce failures accurately, or to induce common failures in order to test new recovery mechanisms. Moreover, the cloud's elastic resource allocation can be used, after the tests activities have completed successfully, to deploy the upgraded system in production by scaling up the test environment. This approach prevents the upgrade failures due to differences between testing and deployment environments [10]. A challenge for the future is to investigate the impact of testing systems in environments that mirror faithfully all the attributes of the deployment environment—except for scale—in order to understand the limitations of such cloud-specific approaches.

3.6. The Need for Upgrade Dependability Benchmarks

Evaluations of upgrade mechanisms focus on assessing performance or the range of updates supported, rather than on dependability. Dependability benchmarking is challenging because, unlike system performance, the dependability attributes (e.g., availability, reliability) cannot be measured directly. A benchmark able to provide quantitative comparisons of the dependability of multiple upgrade mechanisms must be based on a large sample of empirical observations, in order to ensure statistical relevance.

The lack of a standard benchmark for upgrade dependability is the result of the scarcity of data on upgrade failures and, more generally, on the stability of software components in different environments. The failures of software upgrades represent a sensitive subject, which prevents organizations from sharing the information required for replicating these failures outside of the deployment environments. To make progress in this direction, we must establish a comprehensive corpus of realistic faults that commonly occur during online upgrades, collected from multiple industry sources. Similar repositories, such as the top 25 programming errors that lead to security vulnerabilities [50], have had a significant impact on the practice of programming, and a benchmark for software upgrades will likely have a positive impact on the availability of cloud systems.

4. Conclusions

The advent of cloud computing emphasizes the importance of service availability; in fact, for an increasing number of mission-critical applications, availability becomes subject to contractual obligations. We present evidence that the infrastructure that underlies, and the applications that rely upon, cloud computing undergo a fast-paced evolution, which mandates the introduction of online upgrade techniques to avoid service interruptions. We also show that the current techniques for upgrading enterprise systems online are not well suited for the unique technical characteristics and the evolution pace of cloud computing. In particular, techniques, such as rolling upgrades, that have been developed for systems with fixed resources do not take full advantage of the elastic resource-allocation capabilities of cloud computing. These capabilities provide an alternative to rolling upgrades and enable the testing of upgraded systems in an environment that faithfully mirrors the attributes of the deployment setting (except for the scale targeted). Moreover, cloud-based systems are able to ship client-side code to users whenever they connect to the service, which reduces the concerns for the evolution of clients and eliminates many of the technical challenges for upgrading communication protocols.

However, in a cloud environment, data and computation are distributed across multiple nodes, tiers, and even data centers, which poses the risk of upgrade-induced inconsistencies, at multiple levels. The fast evolution pace of cloud software shrinks the opportunities for reasoning about, and eliminating, such inconsistencies, because the concrete evolution steps are often hidden from programmers and the myriad interactions between multiple program versions make it difficult to understand the possible states of the system. We propose adding explicit support for evolution to programming languages and tools, to alleviate this development and analysis burden. To be able to assess the effectiveness of these techniques, we also emphasize the need for establishing a benchmark for the dependability of software upgrade mechanisms, based on field-gathered data on upgrade failures.

Acknowledgments. This research was supported in part by NSF grant CCF-0963996. We thank Greg Ganger and the anonymous referees for their feedback.

References

[1] Ellen Messmer, "U.S. military takes cloud computing to Afghanistan," September 2010, http://www.networkworld.com/news/2010/092310-cloud-computing-afghanistan.html.

[2] Microsoft, "Microsoft Amalga Featured Customers," http://www.microsoft.com/amalga/customers/bysegment.mspx.

[3] Amazon, "AWS Service Health Dashboard," http://status.aws.amazon.com/. Retrieved on 11 May 2011.

[4] S. Lohr, "Amazon's Trouble Raises Cloud Computing Doubts," *The New York Times*, April 22, 2011, http://www.nytimes.com/2011/04/23/technology/23cloud.html.

[5] Amazon, "Summary of the Amazon EC2 and Amazon RDS service disruption in the US East region," http://aws.amazon.com/message/65648/.

[6] Google, "Google Apps service level agreement," 2011, http://www.google.com/apps/intl/en/terms/sla.html.

[7] L. C. Toy, “Large-scale real-time program retrofit methodology in AT&T 5ESS switch,” in *Reliable computer systems: design and evaluation*, 2nd ed., 1992.

[8] B. Malik and D. Scott, “Best practices for continuous application availability,” in *Gartner Data Center Conference*, Las Vegas, NV, Dec 2008.

[9] O. Crameri, N. Knežević, D. Kostić, R. Bianchini, and W. Zwaenepoel, “Staged deployment in Mirage, an integrated software upgrade testing and distribution system.” SOSP’07.

[10] T. Dumitraş and P. Narasimhan, “Why do upgrades fail and what can we do about it? Toward dependable, online upgrades in enterprise systems.” Middleware’09.

[11] ——, “No downtime for data conversions: Rethinking hot upgrades,” Carnegie Mellon University, Tech. Rep. CMU-PDL-09-106, 2009.

[12] E. A. Brewer, “Lessons from giant-scale services,” *IEEE Internet Computing*, 2001.

[13] D. Chappell, “Introducing Windows Azure,” December 2009, sponsored by Microsoft Corporation.

[14] Microsoft Corporation, “Perform a rolling upgrade from Windows 2000,” TechNet Library, Jan 2005. [Online]. Available: http://technet.microsoft.com/en-us/library/cc738005(WS.10).aspx

[15] Oracle Corporation, “Database rolling upgrade using Data Guard SQL Apply,” Maximum Availability Architecture White Paper, Dec 2008. [Online]. Available: http://www.oracle.com/technology/deploy/availability/pdf/maa_wp_10gr2_rollingupgradebestpractices.pdf

[16] M. Segal, “Online software upgrading: new research directions and practical considerations.” COMPSAC’02.

[17] D. Dig and R. Johnson, “How do APIs evolve? A story of refactoring,” *JSME*, 2006.

[18] C. Gkantsidis, T. Karagiannis, and M. Vojnovic, “Planet scale software updates.” SIGCOMM’06.

[19] D. Petrou (Google), Personal communication, 2010.

[20] Facebook, “Testing Using The Beta Tier,” http://developers.facebook.com/blog/post/438.

[21] Sumo Ltd, “Sumo Paint,” http://www.sumopaint.com.

[22] V. Mandalapa, “A framework for understanding schema evolution in web information systems,” Master’s thesis, Arizona State University, 2009.

[23] NCBI, “BLAST,” ftp://ftp.ncbi.nlm.nih.gov/blast/executables/release/.

[24] Roger S. Barga, “Cloud Computing for Research,” March 2011, http://nsfcloud2011.cs.ucsb.edu/.

[25] D. R. Zerbino and E. Birney, “Velvet: Algorithms for de novo short read assembly using de bruijn graphs,” *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.

[26] D. R. Hipp, “Sqlite,” http://www.sqlite.org/.

[27] A. Downing (Oracle Corporation), Personal communication, 2008.

[28] Wikimedia Foundation, “MediaWiki 1.5 upgrade,” 2005. [Online]. Available: http://meta.wikimedia.org/wiki/MediaWiki_1.5_upgrade

[29] C. A. Curino, H. J. Moon, and C. Zaniolo, “Graceful database schema evolution: the prism workbench,” *VLDB*, vol. 1, no. 1, pp. 761–772, 2008.

[30] S. Wu and I. Neamtiu, “Schema evolution analysis for embedded databases,” *HotSWUp’11*.

[31] D.-Y. Lin and I. Neamtiu, “Collateral evolution of applications and databases,” in *EVOL/IWPSE’09*.

[32] F. Schmuck and R. Haskin, “GPFS: A shared-disk file system for large computing clusters,” in *FAST’02*.

[33] F. Schmuck (IBM Research), Personal communication, 2010.

[34] T. Dumitraş, I. Neamtiu, and E. Tilevich, “Report on the second ACM workshop on hot topics in software upgrades (HotSWUp’09),” *SIGOPS OSR 2010*.

[35] T. Dumitraş, E. Tilevich, and P. Narasimhan, “To upgrade or not to upgrade: Impact of online upgrades across multiple administrative domains,” in *ACM SPLASH Onward!*, Oct 2010, pp. 865–876.

[36] M. E. Segal and O. Frieder, “Dynamically updating distributed software: supporting change in uncertain and mistrustful environments,” in *ICSM’89*.

[37] I. Neamtiu and M. Hicks, “Safe and timely updates to multi-threaded programs,” in *PLDI’09*.

[38] P. Bhattacharya and I. Neamtiu, “Dynamic updates for Web and cloud applications,” in *APLWACA’10*.

[39] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, “Links: Web programming without tiers,” in *FMCO’06*.

[40] O. Rütti, P. T. Wojciechowski, and A. Schiper, “Structural and algorithmic issues of dynamic protocol update,” in *IPDPS’06*.

[41] A. Anderson and J. Rathke, “Migrating protocols in multi-threaded message-passing systems,” in *HotSWUp’09*.

[42] J. Waldo, “The end of protocols,” *Sun Microsystems Java Developer Connection*, June 2000. [Online]. Available: http://java.sun.com/developer/technicalArticles/jini/protocols.html

[43] J. Waldo (VMware), Personal communication, 2010.

[44] S. Hanna, E. Chul, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song, “The emperor’s new APIs: On the (in)secure usage of new client-side primitives,” in *W2SP’10*.

[45] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, “Automated detection of refactorings in evolving components,” in *ECOOP’06*.

[46] G. Bierman, M. Parkinson, and J. Noble, “UpgradeJ: Incremental typechecking for class upgrades,” in *ECOOP’08*.

[47] E. Tempero, G. Bierman, J. Noble, and M. Parkinson, “From Java to UpgradeJ: an empirical study,” in *HotSWUp’08*.

[48] C. Hayden, E. Smith, M. Hicks, and J. Foster, “State transfer for clear and efficient runtime upgrades,” in *HotSWUp’11*.

[49] X. Ding, H. Huang, Y. Ruan, A. Shaikh, B. Peterson, and X. Zhang, “Splitter: A proxy-based approach for post-migration testing of Web applications,” in *EuroSys’10*.

[50] CWE/SANS, “Top 25 most dangerous programming errors,” Dec 2010. [Online]. Available: http://cwe.mitre.org/top25/