Dynamic Updates for Web and Cloud Applications

Pamela Bhattacharya Iulian Neamtiu

Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521, USA
{pamelab,neamtiu}@cs.ucr.edu

Abstract

The center of mass for newly-released applications is shifting from traditional, desktop or server programs, toward web and cloud computing applications. This shift is favorable to end-users, but puts additional burden on application developers and service providers. In particular, the newly emerging development methodologies, based on dynamic languages and multi-tier setups, complicate tasks such as verification and require end-to-end, rather than program-local guarantees. Moreover, service providers need to provide continuous service while accommodating the fast evolution pace characteristic of web and cloud applications. A promising approach for providing uninterrupted service while keeping applications up-to-date is to permit dynamic software updates, i.e., applying dynamic patches to running programs. In this paper we focus on safe dynamic updates for web and cloud applications; we point out difficulties associated with dynamic updates for these applications, present some of our preliminary results, and lay out directions for future work.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Reliability, availability, and serviceability; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Corrections, Enhancement; F.3.2 [*Semantics of Programming Languages*]: Program analysis

General Terms Reliability, Verification

Keywords Web applications, cloud computing, dynamic software updating, online updates, on-the-fly upgrades, end-to-end properties

1. Introduction

We are currently witnessing a shift in how applications are developed and deployed. Desktop applications are transitioning to web applications, and server applications are transitioning to "solution stacks" and cloud computing. For example, office suites are moving from local applications to web applications such as Google Docs and Microsoft Office Web Apps. More generally, traditional client-server applications—where the client is "thin" and provides little functionality—are giving way to "thick," feature-rich clients. On the server side, developing applications has recently been facilitated by open source frameworks such as Django, Ruby on Rails, or Google App Engine. Rather than writing C/C++/Java/SQL code for each tier (front, application, database), developers can use one of these frameworks which allow easy software construction and deployment.

The Cloud Computing paradigm has gained popularity for hosting applications in recent years, as it provides businesses with pay-as-you-go computation and storage services. Data infrastructures and applications associated with cloud computing and web applications require regular maintenance and updates (to provide the newest features or incorporate the latest security fixes), while also having to provide 24/7 service. Traditional maintenance and update practices are ill-suited for these environments, because they are based on stop-restart (stopping the system, followed by restart to install updates) or rolling upgrades (where one part of the infrastructure is updated at a time, rendering it unavailable to clients). This temporary unavailability can lead to failures [21], customer dissatisfaction, or even loss of revenue; as service providers use a pay-per-use model for cloud services, unavailability results in loss of business. For example, in 2009, Google offered to compensate users of Google Apps Premier Edition customers for periods of unavailability [25] due to two major Gmail outages [12, 28]; these outages took place during rolling upgrades. Further anecdotal evidence from Facebook and Oracle [13] stresses the need for continuous availability. Therefore, on-the-fly updates are becoming a required feature for maintaining client satisfaction in light of more frequent patches [23] and regular maintenance.

In prior work, we have shown that dynamic updates are effective for Internet servers (such as FTP servers, SSH servers, media streaming and web caching servers [18–20]); in more recent work, we have explored on-the-fly changes

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

APLWACA '10 June 6, Toronto, Canada.

Copyright © 2010 ACM 978-1-60558-913-8/10/06. . . $10.00

to database schemas. However, web applications and cloud computing present unprecedented challenges in software updating in general, and dynamic software updating in particular. Our paper points out how the models underlying these novel applications complicate updates: the dynamic nature of the applications makes verification tasks difficult, and their distributed architecture raises consistency issues. We discuss these challenges, point out how state-of-the-art solutions fail to completely address them, and present our preliminary results on two fronts: (1) providing end-to-end dynamic update mechanisms, and (2) identifying and eliminating update safety issues due to cross-tier and cloud-wide inconsistencies.

2. Challenges And State Of The Art

2.1 Dynamic Languages

The advent of Web 2.0 and the concept of Web as a "participation platform" gave the users more interactivity than just retrieving information, by allowing them to run software applications entirely through a browser. This paradigm led to the popularity of many dynamic scripting languages, such as JavaScript, Python, and PHP, and frameworks like Django or Ruby on Rails, that permit either direct generation or facilitate the construction, of multi-tier software. The evidence for this shift is not merely anecdotal: language popularity statistics[1] show that the number of lines of code written for new software, in dynamic languages, is comparable to that of C, C++, and Java. Similarly, the Tiobe index,[2] a measure of language popularity, shows that, as of March 2010, dynamic languages (e.g., Python, JavaScript, Ruby) are gaining popularity, whereas C, C++, and Java are losing ground.

While these dynamic languages and frameworks based on them enable rapid construction of complex web applications, they introduce two main hurdles to safe dynamic updates:

1. Since these new languages are dynamic, the lack of static checking and the lack of mature analysis and verification tools makes them more prone to error. Moreover, as applications increase in size, they become increasingly difficult to maintain.
2. Multi-tier applications require end-to-end verification and property enforcement (e.g., security), as opposed to local guarantees associated with monolithic applications.

Our current and future efforts tackle these issues by leveraging emerging verification tools for dynamic languages and formal ways of ensuring end-to-end properties [9, 15]. Preliminary results, by us and other researchers suggest that (1) designing or incorporating static checking into dynamic scripting languages are effective verification mechanisms [5, 14, 22, 24, 27], and (2) formally modeling the semantics of each tier in multi-tier applications and checking cross-tier consistency helps guarantee end-to-end properties for the whole application [8, 10, 11, 17].

2.2 Cloud Computing

Cloud Computing is evolving as a powerful paradigm for hosting Internet-scale applications in large computing infrastructures. In particular, IT services are migrating from enterprise-scale computing infrastructures (i.e., in-house networked cluster of servers) to cloud computing infrastructures (i.e., pay-for-service large data-centers with thousands to tens of thousands of machines). The pay-per-use model, zero up-front investment, and perception of unlimited resources/infinite scalability, are major features attracting a large group of users [2, 3]. On the flip side, applications based on cloud computing must provide continuous availability—paying clients can not tolerate downtime associated with system reboot to perform updates—hence fast, on-the-fly, updates are a necessity. Therefore, traditional maintenance and updates practices based on stop/restart or rolling upgrades become ill-suited in this environment. In particular, rolling upgrades, despite their prevalence, are problematic because the upgrade is not an atomic operation and it risks introducing inconsistencies in the application stack, as illustrated in Section 2.4.

2.3 Update Mechanisms

Prior work by us and others [18–20] has shown that on-the-fly software update mechanisms are practical for standalone applications. Extending these mechanisms to web and cloud applications is not trivial, due to several factors. In particular, dynamic typing and dynamic linking (in contrast to statically-typed and linked C/C++/Java applications) increase flexibility, but decrease the testing and verification capacity. Similarly, dynamically generated code (in contrast to the manually-written code characteristic of C/C++/Java applications), open the way for cross-tier inconsistencies and update failures.

2.4 Update Safety

Prior to Web 2.0, the thin client model was prevalent: most of the computation took place on the server, and the client was relegated to HTML rendering. Therefore, in that model, verifying the application logic, reasoning about consistency issues, and enforcing safety properties for server programs alone was sufficient. In Web 2.0, the application model is based on thick clients supported by feature-rich browsers; computation is offloaded to the client, which communicates with the server more sparsely. In cloud computing with multi-tenant architectures, a single application instance partitions its data and configuration to provide different, custom interfaces to different client organizations. Thus, in these novel models, delineating data associated with one connection or one client is difficult— the multi-tier or multi-tenant

[1] `http://www.langpop.com/`

[2] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

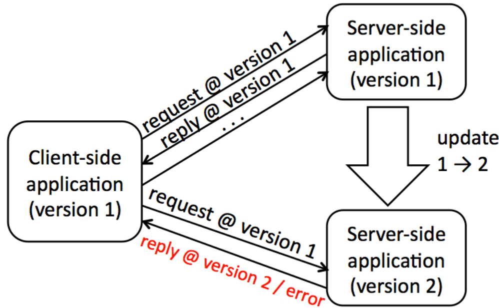

Figure 1. Inconsistencies due to long-running clients and server-side updates.

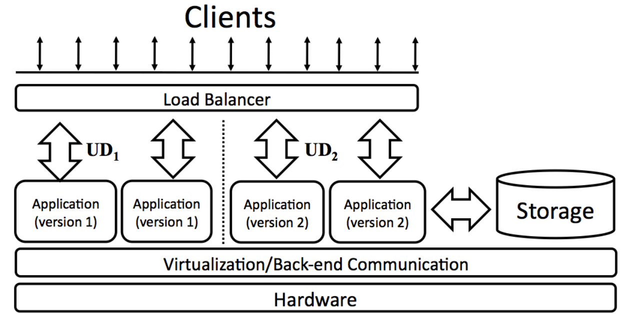

Figure 2. Multiple application versions as a result of rolling upgrades in the cloud.

setup transforms the need for local property checking into the need for end-to-end property checking.

For example, many web applications include plugins or even embedded sub-applications that personalize larger, container applications according to user needs—in Gmail, an email message which includes keywords for appointments automatically gives the user the option to add the message to the calendar. Though convenient, plugins and embedded applications raise security issues which can undermine the security of the *entire* container application. Therefore, guaranteeing application security requires the consideration of all components and across all tiers, i.e., *end-to-end consistency*.

Programming models and frameworks such as Links [10], Swift [8], and Google Web Toolkit [3] address this problem by allowing developers to write a single application, and letting the compiler generate separate code for each tier while preserving consistency. However, even when using this model, updates are problematic, e.g., due to long-lived clients or rolling upgrades, as illustrated by the following examples.

Unsafe updates due to long-lived clients. Long-lived clients whose sessions last longer than the interval between server-side upgrades can lead to inconsistent updates, as illustrated in Figure 1. In this scenario, the client-side application starts its communication with the server at version 1. At some point, the server-side application undergoes an update from version 1 to version 2. Any post-update communication is potentially unsafe, as the client-side part of the application is now out of sync (i.e., still at version 1). Adding versioning to client-server messages addresses this problem, but anecdotal evidence suggests that, in practice, versioning is too complicated and unpopular with service providers [13]. As a consequence, service providers prefer the simplicity of non-versioning approaches (at the expense of safety), or permit addition-only updates which alleviate safety concerns. However, even in the presence of versioning, updating thick-client applications is problematic, due to client-stored persistent data; one such example is Gears,[4] a framework used in popular web applications such as Google Docs and Gmail. Gears provides caching, storage and parallel execution functionality by extending the client's browser. Schema updates, e.g., as a result of a new application version, can lead to the database schema assumed by the web application and the database schema assumed by local storage to differ, which is potentially problematic.

Unsafe updates due to server-side rolling upgrades. Update safety is further threatened in cloud computing applications. Rolling upgrades allow service providers to sustain service yet perform upgrades by partitioning the server-side applications into "update domains." We illustrate rolling upgrades in a cloud computing environment in Figure 2: business logic applications run on top of virtualized hardware, and communicate with clients (via a load balancer) and cloud storage. Prior to applying an update from version 1 to version 2, cloud service providers partition the applications into two update domains, $\mathbf{UD_1}$ and $\mathbf{UD_2}$. Systems in update domain $\mathbf{UD_1}$ will continue serving clients at version 1, while systems in update domain $\mathbf{UD_2}$ wait for client requests to complete, shut down and restart at version 2. The result (Figure 2) is that the $\mathbf{UD_1}$ partition runs application version 1, while the remaining applications run version 2. This situation is problematic when application instances communicate with other instances (e.g., when an instance from $\mathbf{UD_1}$ requests a field deleted in version 2 from an application instance in $\mathbf{UD_2}$) or with the storage system (e.g., when an application instance-assumed data schema differs from the schema assumed by the storage services). Note that Microsoft Azure [6, 7] employs rolling upgrades and allows different application versions to run in parallel; rolling upgrades at Google have led to Gmail outages [12, 28].

3. Preliminary Results and Future Work

We now proceed to presenting our current work and preliminary results on *update mechanisms*, i.e., extending our local dynamic update mechanisms to all links in an end-to-

[3] `http://code.google.com/webtoolkit/`

[4] `http://gears.google.com/`

end chain, and *update safety*, i.e., ensuring a uniform semantics across tiers and across cloud-side applications. Together, these two results lead to safe, end-to-end dynamic updates.

3.1 Update Mechanisms

Preliminary results. In prior work, we have demonstrated a practical approach for performing on-the-fly updates to popular server programs written in C (*memcached*, FTP and SSH servers, routers, media streaming servers, online gaming servers [18–20]); other researchers have shown similar results for dynamic updates to Java applications [26]. All these updates could be applied while sustaining service to clients. These solutions have proved appropriate for server applications without persistent state, but inadequate for applications that use databases—as applications evolve, so do the schemas at which they store data, so we must support on-the-fly schema changes as well. Therefore, in more recent work we have started to explore the feasibility of dynamic updates to persistent data, via on-the-fly schema evolution. Initial results on SQLite-based systems[5] using actual schema changes from Mozilla show that database applications can enjoy safe schema updates with little performance cost.

Future work. Our existing techniques place us part way there in offering dynamic update mechanisms to multi-tier applications, i.e., we can perform updates to the database tier and the application tier (as long as the application tier is written in C or Java). We are, however, missing update mechanisms for dynamic languages (e.g., PHP, Python, or Ruby), so one of the next steps is to add on-the-fly update support to dynamic languages, inspired by existing support in languages such as Erlang [4]. We are also planning to investigate dynamic updates to client code—these are necessary in the presence of thick clients as indicated in Section 1—hence completing the end-to-end dynamic update chain. To guarantee update safety, update mechanisms to links in the end-to-end chain will need to preserve cross-tier and cloud-wide consistency, aspects we will focus on next.

3.2 Update Safety

Preliminary results. In Section 2.4 we discussed the challenges of maintaining cross-tier consistency in an application in the absence of evolution. In prior work [17], we studied cross-tier consistency issues that occur at the application/database boundary when the database or the application, or both, evolve. We have shown that, as a result of software evolution (i.e., a new version of the application), the application tier and the database tier could make different assumptions about the database schema. This could potentially lead to runtime errors, data loss, and loss of data integrity. Our study on several years of evolution in two popular open source programs, Mozilla and Monotone, has found that, for certain versions, the application can be out of sync with the database because they assume different schemas. In that work we manually compared the application-assumed and database-assumed schemas, and showed how they can differ as a result of new releases. Motivated by these initial findings, we plan to (1) extend the work to a wider range of cross-tier consistency issues, and (2) make the approach more scalable, by leveraging existing tools for automatic model extraction and cross-tier verification.

[5] SQLite is a popular, server-less, zero-config SQL engine [16].

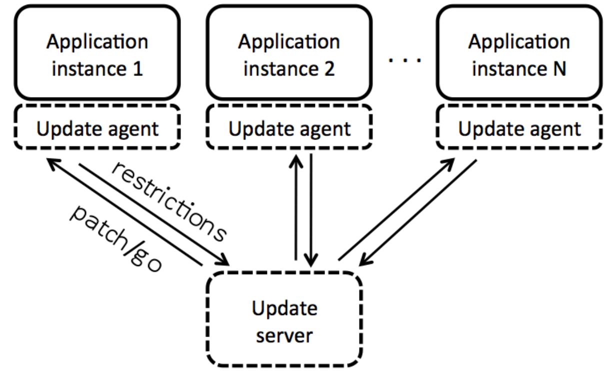

Figure 3. Cloud-wide dynamic updates to applications.

Future work: cross-tier safety. The first step is to use an automatic approach for extracting the application model (e.g., types, or the state machine that describes application logic, or the database schema) assumed by each tier, from the JavaScript client to Python or Ruby code on the server, to the server-side database. The second step will be to verify and guarantee the safety of manual and automatically generated code, by leveraging several existing static analysis and profiling tools for dynamic languages, including Ruby [14], JavaScript [22, 27], and Python [5, 24]. The third step is to take local properties for each tier (e.g., security guarantees), and check and enforce those properties for the client-server chain as a whole, effectively providing end-to-end guarantees.

Future work: cloud-wide application update safety. For implementing safe, cloud-wide updates without relying on rolling upgrades, we plan to use a combination of static and dynamic techniques, similar to the approaches employed by us and other researchers in the context of multi-threaded [18] and distributed [1] updates. We illustrate the high-level principle in Figure 3. To allow safe updates to all application instances $1..N$ at once, and obviate the need for rolling upgrades, we need to apply the update at a moment where the update contents ("patch") does not conflict with the current state of any application instance. Therefore, each application instance is augmented with an *update agent* whose role is to communicate with an *update server*. Note that the agents and server do not exist in current cloud setups and will need to be added. The job of an update agent is to communicate the local update restrictions of each instance to the update server, and to install the patch when it receives the "go" signal from the server. The update server centralizes update re-

strictions, finds a moment when applying an update is safe across-the-board (i.e., the update does not violate the update restrictions of any application instance), and sends update agents the "go" signal.

We plan to use *contextual effects* [20] and *relaxed synchronization* [18]—a combination of static and dynamic approaches—to compute dynamic update restrictions for each application instance; the update server will check these restrictions against update contents to detect safe system states in which an update can be applied on-the-fly for all application instances. Our preliminary experience with using relaxed synchronization for applying dynamic updates to multi-threaded programs with large numbers of threads gives us confidence that the approach can be translated to a cloud setup to achieve cloud-wide, safe dynamic updates.

4. Conclusions

As Web 2.0 and cloud services are gaining popularity, the computing paradigm is shifting from large monolithic applications to thick clients, solution-stack based servers, and cloud-based computation and storage services. These novel applications depend on continuous server availability, which puts pressure on service providers to keep their services running 24/7 and employ on-the-fly updates. Our paper points out that the models underlying these novel applications complicate on-the-fly updates: the dynamic nature of the applications makes verification tasks difficult, and their distributed architecture opens the way for update-induced inconsistencies. To address these issues, we present our preliminary results and intended future work on dynamic update mechanisms and dynamic update safety for web and cloud applications.

Acknowledgments We would like to thank the anonymous referees for their helpful comments on this paper. This research was supported in part by a UC Regents' Faculty Fellowship.

References

[1] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *ECOOP'06*, pages 452–476.

[2] Amazon. Amazon Web Services provider case studies, 2009. `http://aws.amazon.com/solutions/solution-providers/`.

[3] Amazon. Amazon Web Services user case studies, 2009. `http://aws.amazon.com/solutions/case-studies/`.

[4] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International Ltd., 1996.

[5] B. Cannon. Localized type inference of atomic types in python. Master's thesis, California Polytechnic State University, San Luis Obispo, 2005.

[6] D. Chappell. Introducing Windows Azure. December 2009. Sponsored by Microsoft Corporation.

[7] D. Chappell. Introducing the Windows Azure Platform. December 2009. Sponsored by Microsoft Corporation.

[8] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *SOSP'07*, pages 31–44.

[9] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *PLDI '09*, pages 50–62.

[10] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO'06*.

[11] B. J. Corcoran, N. Swamy, and M. Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD'09*, pages 269–282.

[12] A. Cruz. Update on today's gmail outage, Feb 2009. `http://gmailblog.blogspot.com/2009/02/update-on-todays-gmail-outage.html`.

[13] T. Dumitraş, I. Neamtiu, and E. Tilevich. Second ACM Workshop on Hot Topics in Software UPpgrades (HotSWUp 2009). In *OOPSLA '09*, pages 705–706.

[14] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. The ruby intermediate language. In *DLS '09*, pages 89–98.

[15] S. Guarnieri and B. Livshits. Gulfstream: Incremental static analysis for streaming JavaScript applications. In *USENIX Conference on Web Application Development*, June 2010.

[16] D. R. Hipp. Sqlite. `http://www.sqlite.org/`.

[17] D.-Y. Lin and I. Neamtiu. Collateral evolution of applications and databases. In *IWPSE'09*, pages 31–40.

[18] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. In *PLDI'09*, pages 13–24.

[19] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI'06*, pages 72–83.

[20] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL*, pages 37–49, 2008.

[21] D. L. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USITS'03*.

[22] C. A. Paola, C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *ECOOP'05*, pages 429–452.

[23] E. Rescorla. Security holes... who cares? In *USENIX Security Symposium*, August 2003.

[24] M. Salib. Starkiller: A static type inferencer and compiler for python. Master's thesis, MIT, 2004.

[25] S. Shankland. CNET News: Google Gmail outage compensation: $2.05 per user, 2009. `http://news.cnet.com/google-gmail-outage-compensation-$2.05-per-user/`.

[26] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *PLDI'09*, pages 1–12.

[27] P. Thiemann. Towards a type system for analyzing javascript programs. In *European Symposium On Programming*, 2005.

[28] B. Treynor. More on today's gmail issue, Sept 2009. `http://gmailblog.blogspot.com/2009/09/more-on-todays-gmail-issue.html`.