Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.18576/amis/100215

A Performance-based Approach to Automatic Redeployment of Enterprise Software Applications

Habib Seifzadeh[1,*], *Hassan Abolhassani*[2] *and Mohsen Sadighi Moshkenani*[3]

[1] Faculty of Computer Engineering, Najafabad Branch, Islamic Azad University, Najafabad, Iran
[2] Computer Engineering Department, Sharif University of Technology, Tehran, Iran
[3] School of Science and Engineering, Sharif University of Technology, International Campus, Kish Island, Iran

Received: 12 Aug. 2014, Revised: 7 Jul. 2015, Accepted: 16 Aug. 2015
Published online: 1 Mar. 2016

Abstract: Update of software usually equates disruption to a program's execution. However, such disruptions in the execution of applications that provide round-the-clock services is not desirable. Examples of such applications include multi-tier enterprise systems with which users may interact via distinct presentation tiers at the same time. Existing software updating systems do not operate in the enterprise applications, or they need most of the redeployment tasks to be performed manually. In this paper, we present a framework capable of Automatically Redeploying Enterprise Software Applications (ARESA). This system combines the idea of incremental and integral updates to preserve the consistency of applications and to minimize disruption times during redeployments at the same time. It also utilizes the update bringing forward techniques in order to enhance the system's predictability. ARESA has been used in updating a *desktop costing* application, a *custom web* program, and *Joomla* content management system. The experimental results show that in applications with real-life complexities, 'speed' and 'size' overheads incurred by ARESA are less than 19% and 1%, respectively.

Keywords: Software Maintenance, Dynamic Software Updating, Enterpirse Software Systems, Availability

1 Introduction

Traditional redeployment[1] of multi-tier enterprise software systems requires all the tiers to be stopped and replaced with their new versions so that the execution could be continued. This process is labour-intensive for the programmers and it also causes long disruptions for the end-users. Today, most highly available applications such as industrial web sites, project and content management systems, transactional banking and embedded control applications suffer from these unpleasant disruption times.

Dynamic Software Updating (DSU) systems are used to update software applications at run-time in order to reduce disruptions experienced by the end-users [1–8]. Existing dynamic updating systems are usually only able to update particular tiers of the applications, and therefore, are not suitable for redeploying enterprise software systems [2, 5, 7, 9–11]. Besides, some researchers have claimed dynamic updating systems are not capable of simultaneously updating distinct front-ends in the presentation tier [7, 12, 13].

On the other hand, there are facilities in the software maintenance literature simplifying redeployment of the enterprise applications [14–16]. However, most parts of the redeployment tasks must be performed manually in these facilities [14–16]. Therefore, they do not resolve the issue of long execution interruptions in the enterprise systems. Based on this literature, an updating system able to automatically update all tiers of a multi-tier enterprise application in order to minimize the redeployment disruption time has remained an elusive endeavor [2, 5, 7, 9, 13].

The current study is an attempt to automate the redeployment of multi-tier enterprise software systems which may include distinct front-ends in the presentation tier. To do so, ARESA provides a novel update model that combines integral updates, which replace the whole application at run-time, with incremental updates that

[1] We mainly use the terms 'redeployment' and 'update' for the whole process of updating an enterprise application and upgrading a special component respectively, although they may even be used interchangeably.

* Corresponding author e-mail: seifzadeh@iaun.ac.ir

replace only modified parts. This is to ensure consistency while reducing the end-users' short disruptions. Another unique feature of ARESA is its update timing approach which is a cross between the programmer-specified and the update bringing forward techniques, and this makes it more predictable [13]. Section 4 describes the approaches used in ARESA precisely.

With respect to the implementation, ARESA has been developed as web services in order to simplify its distribution and extension. It currently supports updating of applications written in Java, Microsoft C# and PHP programming languages alongside MySQL and Microsoft SQL Server database management systems. In addition, the following by-products have been obtained during development of ARESA: (1) DBComparator which reports differences of two databases facilitating generation of the database patches, (2) DSUInjector which injects automatic updating facilities to the source code of a front-end to make it dynamically updatable, and (3) DBFiller that populates a database with desired number of temporary records for performance testing of our proof of concept framework.

ARESA has been evaluated on: (1) HABC, a costing application developed in Microsoft C# 2008 and Microsoft SQL server 2005 which performs heavy costing calculations, (2) a custom web application developed in PHP 5.3 and MySQL[1] 5.5 that fetches and displays large amount of information from the database, and (3) Joomla[2], a popular *Content Management System* (CMS) deployed on Apache[3] 2.2 and MySQL 5.5. Performance barely degrades with ARESA deployed, at most 15% in popular web-based applications such as Joomla, and no more than 19% in startup of real-life desktop programs[4]. Also, experiments show ARESA does not impose more than 1% overhead on the size of conventional applications.

The remaining sections of this paper are organized as follows: Section 2 reviews the software updating literature and describes the goals of ARESA in detail. Section 3 includes the description of applications for which ARESA has been designed and the assumptions made in this paper. In Section 4, the leveraged approaches used in this paper are described. In Section 5, we present ARESA's architecture and its implementation details as well as the process of patch generation and application. Sections 6 and 7 provide correctness and practicability proof of ARESA, respectively. The paper concludes with Section 8, providing a summary of the work and areas of future or further work.

Main contributions of ARESA in comparison with the currently available software updating systems are:

–ARESA automates redeployment of the enterprise software systems composed of multiple tiers which may have distinct front-ends in the presentation tier,
–ARESA provides a novel update model and an update timing technique to achieve short disruptions, predictability and consistency at the same time,
–ARESA supports state transformation of all tiers during the redeployment, such as in case of database or web front-end upgrades.

2 Background and related work

As described in Section 1, the main goal of ARESA is to reduce disruption times during the redeployment of the multi-tier enterprise software applications. In this section, we first introduce the multi-tier enterprise applications. Then, the studies conducted on the maintenance of such systems are reviewed and discussed. Finally, the exact definition of the problem ARESA has been designed to tackle is given.

Multi-tier enterprise software applications are systems which are usually composed of three distinct tiers. The presentation tier is used to get inputs from the end-users and send outputs to them. This tier contains one or possibly more front-ends (e.g., web, desktop, mobile) to present the users with results in different formats. The application or logic tier encompasses business workflows and the data tier includes a database as well as components through which the other tiers can communicate with the database. Upgrading such an application requires components of all tiers to be replaced with their new versions.

One way to update the enterprise software applications is to use redeployment facilities provided by the enterprise execution platforms [14–16]. Although these facilities enjoy the advantage of consistent redeployment which is due to the atomic performance of redeployments, they have rarely been devised for certain execution platforms such as PHP. Furthermore, these facilities mainly do not automate the whole process of the redeployment, and therefore, they must be integrated with other manual tasks in order for the redeployment to become complete [14–16]. This is not in line with our short disruption time objective.

Another method of updating an enterprise software application is to leverage dynamic software updating systems [1, 13]. These systems are able to update programs at run-time without the end-user's intervention. Most of the dynamic updating systems also transfer the state of previous programs during the upgrades so the end-users do not have to re-submit their uncompleted tasks after upgrades [1, 2, 4, 6, 17–19]. This causes the update disruption experienced by the end-users to be even more shortened.

To our knowledge, the concept of dynamic software updating has emerged from the early research studies on the software maintenance such as Fabry, and

[1] http://www.mysql.com
[2] http://www.joomla.org
[3] http://www.apache.org
[4] Application clients, programs or front-ends are used interchangeably throughout this paper.

DYMOS [20, 21]. Those systems are mostly able to dynamically update desktop applications written by special-purpose programming languages developed at those times. Other early DSU systems dynamically update components of the operating systems [22]. Because of the importance of availability in the operating systems, the researchers have been conducting studies focusing on the dynamic updating of the operating systems ever since [23–27].

There are other dynamic updating systems able to dynamically update embedded and real-time applications [8, 28–31], desktop applications [4, 17, 32–35], server programs [1, 2, 18, 36–40], and components of distributed systems [19, 41–43]. The above on-line updating systems are only able to update codes in the application logic tier at run-time. The other group of dynamic updating systems deal with upgrading the data tier [9, 44, 45]. To this end, they upgrade databases without propagating data modifications to the other tiers of the application. Although these systems manage to update the databases at run-time, they fail to upgrade the application logic tier or front-ends of the presentation tier.

As the body of research on the dynamic software updating suggests, the currently available DSU systems mainly focus on a particular tier of the application. Due to the complexities in dynamic updating of multiple components of an application, such systems are not well suited for updating multi-tier enterprise software systems. This issue has been identified in several studies [2, 5, 7, 12, 13], and it seems it has remained to be unresolved so far [10, 11, 46]. The issue has become even more noticeable in the enterprise applications consisting of web front-ends, because this type of front-ends have received less attention by the DSU systems compared to their desktop equivalents while they usually have more stringent availability requirements.

Moreover, one of the important problems the DSU systems are dealing with is that they either do not guarantee the consistency completely because of the variety of component versions in the memory [4, 47–49], or in order to preserve consistency they wait too long for the appropriate update time and this makes them unpredictable [1, 6, 25, 34, 36, 37, 50]. Although, a number of studies focusing on simultaneous achieving of both consistency and predictability have been conducted, they do not specifically turn to the enterprise software systems [51, 52].

Considering the above shortcomings in the existing software updating systems, our purpose in this study is to develop an updating system with characteristics listed below: (1) our system should be able to update the entire tiers of the enterprise applications automatically (even those containing distinct front-ends in the presentation tier), (2) it should redeploy applications within short and predictable time frames, (3) it must not violate programs' consistency because of the updating, and (4) it has to transfer the state of all tiers to the new versions. It should be noted that the consistency will be our main priority in case we fail to keep a balance amongst the above objectives.

3 Assumptions

Designing ARESA, we have made two assumptions explained hereafter:

– *Data-centric applications.* First, we suppose different front-ends of the same enterprise application communicate only via the global database; no direct messages pass back and forth between them. Since ARESA employs the incremental updates under special circumstances, there may be front-ends of different versions in the memory at the same time. This assumption conservatively forces front-ends of distinct versions not to interact with each other directly. To our knowledge, most enterprise applications, especially those using *Model-View-Controller* (MVC) architecture, make similar assumptions. For example, email servers use a single database while their different mail clients such as web mails or *Internet Message Access Protocol* (IMAP) applications access emails only through that database. The other example is banking systems in which all web and mobile applications connect only to a database to perform accounting transactions. Moreover, the above assumption does not restrict ARESA to updating of large number of software applications which have a single user interface alongside their databases. Examples of these systems include CMSs, web project management and collaboration systems.

– *Software requirements.* Since ARESA components have been implemented as Java web services, each machine containing a component of the ARESA must have a Java web container installed and running. Fortunately, due to the multi-platform availability of Java, this requirement does not violate the ARESA generality. This web container can be as small as a Tomcat server[1] occupying a few megabytes of storage, although we exploit the larger weblogic server[2] for our evaluations.

4 ARESA approaches

In this section, we present approaches employed by ARESA in order to fulfill the aims mentioned in Section 2. The ARESA update model has been described in the first two subsections; in the first subsection, it is assumed that only programs of the presentation tier have been modified while in the second subsection, the

[1] http://tomcat.apache.org/

[2] http://www.oracle.com/technetwork/middleware/weblogic/index.html

assumption is that the data tier has been changed. In the third and fourth subsections, the timing and the state transformation techniques used in ARESA are described, respectively. Justification for each methodology is provided along with a discussion of alternative approaches in each case.

4.1 Updating application clients

There are two significant approaches to update programs in the presentation tier in the literature [13]: (1) an incremental model in which only updated components are replaced in the memory [1, 2, 4, 17, 18, 27, 35, 37], and (2) an integral model that replaces the whole programs irrespective of their updated components [36, 40]. The first method prevents unnecessary replacements during the upgrade, resulting in short disruption times [13]. However, this technique interposes an extra level of access indirection between each two components in the program which leads to more overhead on the performance of the programs [13]. Because of the variety of tiers and components in the enterprise software systems, any imposed overhead amongst components may cause significant overall performance degradations. In addition, replacing only modified parts of the application causes components of different versions to co-exist in the memory which may result in consistency violations [13].

The second model, on the other hand, does not require consistency preservation mechanisms because it replaces the whole application at run-time [13]. The other benefit of this approach is that it supports most kinds of changes in the source codes of the front-ends [13]. Nevertheless, this approach causes long update durations owing to the unnecessary module replacements in the memory. Replacing all tiers of the application and their components for small changes in a front-end leads to many unnecessary replacements and is not justifiable in the enterprise systems.

To satisfy our both short disruption time and consistency goals, ARESA combines the two update models described above. It replaces the entire modified front-end while allowing other front-ends and other tiers to continue execution. For instance, consider a banking application with two front-ends, a web-based and a desktop in its presentation tier, and a new version be available for the web front-end to fix a bug in one of its HTML files. ARESA replaces the whole web front-end while it allows the desktop program to continue its execution.

Based on this approach, there is no need to use access indirections within front-ends because the whole modified front-end is renewed. Furthermore, we have a short disruption time because only the modified front-end is redeployed. Of note, this approach causes unnecessary replacements only in the modified front-end which is negligible compared to the overheads on frequent access indirections which have far more negative impact. With regard to the consistency, it is maintained inside the modified front-end because all parts of it are replaced atomically. The consistency is also ensured between the front-ends since the application is data-centric and there is no direct communication amongst front-ends.

4.2 Updating data tier

Based on our empirical study of (1) HABC, (2) Joomla, and (3) *Open Conference Systems* (OCS)[1], database updates are needed in 45% to 75% of applications upgrades. Additionally, a conducted survey indicates that nearly 60% of organizations around the world may have changed their databases in the years 2010 and 2011 [53]. Therefore, supporting the automatic data upgrade during redeployments seems to be essential to reach our short disruption time goal in more than fifty percent of upgrades. To update databases, we are provided with two alternatives: (1) leveraging an extra level of access indirection between the database and other tiers of application in order to prevent database updates from being propagated [9], and (2) redeploying the entire application in the case of database changes.

The first model has a major negative impact on the application's performance since all database queries should be passed through a query rewriting engine. Nevertheless, unlike updating front-ends which does not necessarily result in updating databases (e.g., fixing a bug in a function, improving front-end performance, changing user interface, etc.), we believe that database updates must be reflected in all programs of the presentation tier. For example, if a column is added to a database table, all front-ends should also be updated to utilize the added column in the database, or else the added column will be meaningless or may cause inconsistencies. Therefore, ARESA employs the latter approach to update the databases. Based on this approach, ARESA pauses all front-ends, updates database, and then updates the front-ends with the proper states loaded. In other words, we sacrifice short disruption time for performance and consistency in the case of data tier upgrades.

4.3 Redeployment timing

Choosing an appropriate time of update is a challenging issue in the dynamic updating systems. Several researchers have come to believe that the update timing is an undecidable problem [1, 2, 6, 38, 54–56]. There are numerous approaches to find the appropriate time of upgrade in the literature. Some systems apply updates immediately, either by replacing the active functions [26, 38, 40], or by continuing the execution of

[1] A web-based conference management application, http://pkp.sfu.ca/ocs/

old functions and invoking their new versions afterwards [4, 32, 47]. The former technique is difficult to implement while the latter may jeopardize the applications' consistency, because it causes components of different versions to invoke each other during the upgrades [13].

The other on-line updating systems defer the upgrade until a specific criterion is satisfied. This criterion can be programmer-specified [1, 2, 6, 35], fulfilled when no modified function is active [25, 27, 34, 36, 37], a timeout [8, 57], or quiescence (i.e., no modified function participates in any active transaction of the program) [23, 51]. Systems which use the 'programmer-specified' technique usually suggest programmers choose the end of main infinite loop as the program's update point.

According to the update timing undecidability [54], the 'timeout' and 'no-updated-function-is-active' methods do not completely guarantee the consistency of the programs [13]. The other two methods, 'programmer-specified' and 'quiescence' suffer from unpredictability, because it is not exactly known when the execution reaches the point specified by the programmer or all modified functions finish the execution in the active transactions. The situation worsens in the multi-threaded and distributed applications, because all threads or programs of such applications must converge to the specified points until the update can be applied [38, 52, 58]. Since the enterprise applications are composed of multiple running components, the problem of update point convergence also exists in these systems.

To reduce the aforementioned timing problems, ARESA integrates the programmer time determination with two other update bringing forward mechanisms: (1) timeout and (2) access denial [13]. Determining the appropriate update time by the programmer helps ARESA to be consistent in diverse enterprise systems. For example, in a highly available costing application with multiple front-ends, only the programmer can ascertain whether a specific front-end is busy with complex costing calculations and should not be interrupted or it is sitting idle making this a perfect time for dynamic updating. On the other hand, ARESA reduces the chance of deadlocks by safely discarding a program not reached its update point in the specific time frame. It also makes the web front-ends reject any incoming requests during the upgrades to ensure that the current threads of execution finally converge to the specified update points. Concerning the desktop front-ends, it is the programmer's responsibility to modify the program in a way that it does not accept new requests while being upgraded in the current version of ARESA.

4.4 State transformation

Without a state transformation mechanism, a software updating system becomes a stop/start service which does not have considerable advantages over off-line updates. There are two main techniques to transfer the state from old to new version of the application [13]: (1) re-execution [59, 60], and programmer-specified state transformers [1, 2, 6, 33]. According to the first approach, the execution is rolled back to the nearest point in the old version that is equivalent to a point in the new version, and then the new version is executed from there to the point corresponding to the point in the old version where the execution was interrupted. In the second approach, the programmer provides functions called state transformers to be executed during the upgrade in order to transfer the state from old to new version.

The first technique not only needs the source codes to be annotated by the check-points, it also requires the system to analyze the codes in order to discover similarities and differences among them [59, 61]. These are not trivial tasks, and therefore, the first mechanism has been leveraged by a few systems in the dynamic updating literature. The second technique is simpler to implement. It also enables the dynamic updating system to transform the state of every front-end by invoking the provided methods while programmers can populate them with the content of their choosing. However, it requires programmers to provide more components in order to prepare patches for the automatic redeployments.

Because of simplicity and practicability, ARESA uses the programmer-specified approach. To present this technique can be used in the enterprise applications, we categorize windows or pages of front-ends into three classes and describe how the state of each is transferred by this mechanism: (1) *form* windows with which the end-users enter data, (2) *calculation* pages which enables the end-users to view the progress of their submitted calculation, and (3) *report* pages which show the program's data in different formats. These three classes are common between desktop and web front-ends.

In the form pages, it is important to the end-users that upgrades do not destroy their filled data. In this case, the two functions getState and setState are used to get data from and set them to the graphical components of front-ends, respectively. In desktop programs, *Graphical User Interface* (GUI) data are gathered and filled by the components' provided functions and in the web applications, this happens by the Javascript functions which access HTML elements.

The calculation pages do not require the state transformation during upgrade because the programmers usually do not permit applying updates when this type of pages are viewed. The last class of windows, report ones, contain a database query executed at the page's load and display the information stored in the database based on the executed query. The report pages also do not have any user-provided data. Since databases or report queries may be changed by the upgrades, the state of report pages must be renewed to reflect these changes. This can be achieved by re-opening the opened pages automatically in

order to execute their modified database queries; no other state transformation action is required.

In addition to the pages' states, programs of the presentation tier probably have other invisible states which must be transferred from their old to the new versions. These include global variables in the desktop programs or sub-variables of the *session* and *application* variables in the web front-ends. The two functions getState and setState are also able to transfer these values after they transfer the state of the pages. If a transformation is also needed, it can be performed in either the getState or the setState based on the programmer's needs, although the latter is more probable.

4.5 Environment

Today, service oriented architecture and other distributed communication protocols have led to the emergence of enterprise applications comprising of components written in different programming languages [62]. Therefore, an automatic redeployment system designated for the enterprise applications needs to support most programming languages, or else it probably fail to redeploy the whole application. To this aim, ARESA has been implemented as Java web services. Owing to the portability advantage of Java, the components of ARESA can be installed on different platforms and also be invoked by every programming language able to use the web services. Moreover, this implementation environment enables ARESA to apply updates remotely, eliminating the need for physical presence of programmers or technicians in target organizations.

5 ARESA architecture

Detailed description of the overall architecture of ARESA as well as its patch model is presented in this section. Runtime components and their interactions is also presented.

Redeployment of an enterprise application consists of (1) patch generation, and (2) patch application. As part of the patch generation phase, the programmer constructs and places the patch in a patch repository. In applying the patch, the programmer sends the patch URL to the ARESA runtime environment. Upon receiving a patch URL, the ARESA engine notifies all registered front-ends, saves the execution state of each front-end that has been paused timely, patches the application's database (ensuring no front-end connected), and finally refreshes the front-ends with their correct state.

Front-ends which have not reached their update points in the specified time frame must be renewed manually by using the provided *bootstrap* program. This program also registers the client machine with the ARESA runtime environment for future dynamic updates. With the configuration described above, the programmer is simply required to send the URLs of the patches to ARESA runtime environment in supported organizations. All remaining redeployment steps are automated without any interaction required on the part of the users or the programmer.

5.1 Patch generation

An enterprise patch in ARESA includes three components: (1) an updatable desktop front-end, (2) an updatable web program, and (3) a db patch. As upgrading front-ends does not necessarily requires updating the database, the first two components are essential while the third is optional (refer to Subsection 4.2 of Section 4 for more information).

Fig. 1 illustrates the process of patch generation in ARESA. When a new requirement comes in, the programmer modifies source code of the front-ends and uses the provided DSUInjector tool to make them updatable. The updatable code is able to receive the notification of future updates and store its state for later retrieval by subsequent versions of the code. The programmer completes the updatable template code generated by DSUInjector. The more well-structured a front-end is, the more complete the code generated by DSUInjector; hence requiring less manual tweaking of the code by the programmer. Structure of updatable code and the parts may need the programmer's attention are described for desktop and web applications in the following two subsections, respectively. Since web pages usually do not require compiling, web front-end component of the enterprise patch is ready for deployment at this stage of the process. However, desktop program component must be compiled before it can be embedded in the enterprise patch.

Final step of the process is to provide a SQL script in order to update the enterprise database automatically. There are various database differencing tools available (e.g., mysqldiff[1], OpenDBDiff[2], SQL Workbench/J[3], tablediff[4], and DBComparator). Although these tools assist programmers in preparation of database patches, none can fully automate the process. For example, none is able to detect renaming or moving a field in a database. Therefore, in this version of ARESA, the programmer has to compose the SQL scripts of db patches with the help of the aforementioned tools. The enterprise patch is now complete.

5.1.1 Updatable desktop client

The structure of the updatable desktop applications and their differences with the conventional equivalents are

[1] http://www.mysqldiff.org/
[2] http://opendbiff.codeplex.com/
[3] http://www.sql-workbench.net/
[4] http://msdn.microsoft.com/en-us/library/ms162843.aspx

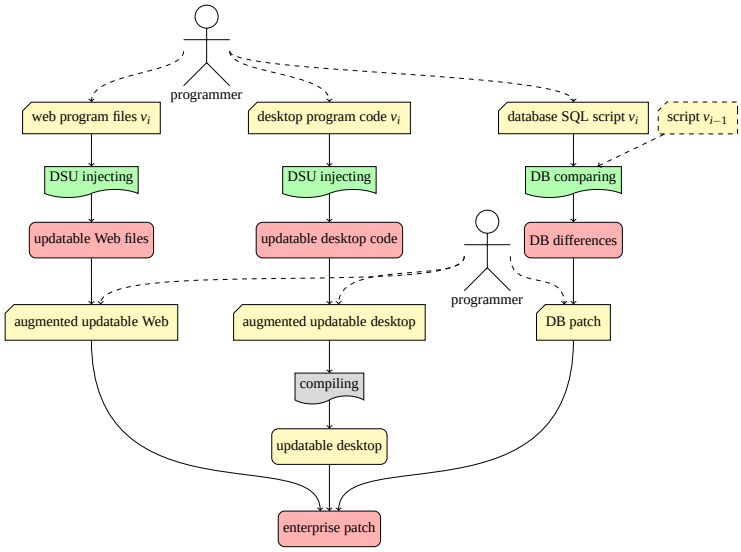


Figure 1: Patch preparation process performed by the programmer

described in this section. This section concludes by a discussion on overheads this model imposes on the space requirement and the execution time of front-ends.

Making a desktop front-end updatable in ARESA requires the following four modifications to be applied to the program's source code. Of note, the first two modifications are made by DSUInjector automatically while the third and fourth should be applied manually at this time.

- –The first modification is to add an instruction to the program's startup routine to create an updater thread. The thread starts by checking whether a previous state has been stored in ARESA runtime environment and if so, loads the state. It then binds to a known port[1] in order to receive the notification of future updates, waits until the execution reaches the nearest update point specified by the programmer, sends the current state to ARESA for subsequent retrieval by the next version of the program, and finally terminates the program in order for the remaining automatic redeployment tasks to be completed.
- –The second modification relates to setters and getters of the program's state. ARESA expects the implementation of two abstract methods, *getState* and *setState*, in DSUInjector at the time of dynamic updating injection. DSUInjector then embeds these two methods into the updatable program in order to be used by the updater thread for state transformation.
- –The third modification is to change hard-coded database connection strings and enable retrieval of such a connection from ARESA. Since developers or even conventional best practices such as Hibernate[2] use *singleton* design pattern [63] for database connection, we make sure only one point is altered in the programs. If it is not the case, the connection string can be requested from ARESA at the application's start-up and be passed to every requiring command during the execution. As an alternative, the connection string can remain hard-coded provided that the programmer sets the name of database's updated version in the connection strings of the new program.
- –The last modification is the update points specified by the programmer throughout the original source code. The more update points the programmer specifies, the shorter disruption the end-user might experience during the redeployment.

A sample redeploying code for Java applications is provided in Fig. 2. In this example, the two abstract methods, setState and getState, are implemented by DSUHandler class. Updater class includes the updater startup thread as well as the DBHandler which contains a method returning the latest database connection string as required. Furthermore, two lines of code have been added to the beginning of the main method in order to create the updater thread and pass an instance of DSUHandler to it. The original main code follows without any changes except for the programmer-specified update points.

As seen in this example, the space overhead incurred by ARESA does not exceed several tens of lines, noting the state setters and getters are provided by the programmer and can be of any length. This is a modest space overhead especially for large front-ends. The time overhead can occur in two places: (1) the updater thread

[1] ARESA uses the *socket programming* to communicate with the desktop programs.

[2] http://www.hibernate.org

```
1  public interface DSURequester {
2    public void setState(State s);
3    public State getState();
4  }
5
6  public class DSUHandler implements
7                          DSURequester {
8    ...//Filled by the programmer
9  }
10
11 public class DBHandler {
12   ARESAClientUpdateService s = new ...;
13   public String getConnString() {
14     return s.getDBConnString();
15   }
16 }
17
18 public class Main {
19   public static void main(...) {
20     DSUHandler h = new DSUHandler();
21     Updater updater = new Updater(h);
22     ... //The original main
23     //Update point specified by programmer
24     if(updater.updateAvailable) {
25       updater.update();
26     }
27     ...
28   }
29 }
```

(a) additions to the original source code

```
1  public class Updater implements
2                              Runnable {
3    DSURequester r;
4    ARESAClientUpdateService s = new ...;
5    boolean updateAvailable = false;
6    ...
7    public Updater(DSURequester r) {
8      ...
9      r.setState(s.getState());
10     this.r = r;
11     new Thread(this).start();
12   }
13   public void update() {
14     s.setState(r.getState());
15     System.exit(0);
16   }
17   public void run() {
18     ...
19     for(;;) {
20       Socket sock = server.accept();
21       String req = readFrom(sock);
22       if (req.equals(UPDATE)) {
23         updateAvailable = true;
24       } else if ... {
25         ...
26       }
27     }
28   }
29 }
```

(b) the updater thread

Figure 2: The source code of an updatable desktop application

execution, and (2) obtaining the new database connection string from ARESA. Once the updater thread is bound to its port, it is merely blocked by the OS awaiting an incoming update notification with minimal time overhead. Assuming the host program utilizes singleton pattern to connect to the database, the majority of the time overhead occur at the program's startup. Overheads of ARESA in several specific applications are discussed in Section 7.

5.1.2 Updatable web client

Despise similarities of the web and the desktop programs, the distributed nature of the web programs makes them more difficult to be automatically updated. Multiple users connected to a single application cause the application's state to be split into two parts of the server side and the client side. This means reaching an agreement on the correct update time or transferring state of the application is not a trivial task. In the following, we explain modifications required to make web front-ends updatable and this brings us to addressing the issues of upgrading these types of programs and the approaches taken by ARESA to tackle them.

In order to make a web front-end updatable, four modifications are required to be applied to the source code. The first and second are performed automatically by DSUInjector and the last two have to be applied manually in the current version of ARESA:

– The first modification is to incorporate a web page called updater[1] into the web program. Requests to this page belong to two main categories, (1) those coming from the front-end's pages, and (2) others sent by ARESA engine[2]. With regard to the former category, this page keeps track of opened pages in the users' browsers, notifies them of new updates, saves state of the opened pages, and sends the stored states back to their new version. With respect to the second category, the updater gets update notification from the ARESA engine, removes web pages that generate

[1] This page has been implemented in PHP and is being developed in other web programming languages.

[2] ARESA utilizes the HTTP requests to interact with the web front-ends.

output in order to refuse any new requests during the upgrade, sends the acknowledgment back to ARESA when all pages save their states meaning they are ready for update, receives the resume message from the ARESA engine and asks pages to redirect to the new versions. It is noteworthy that only one instance of the updater is added to each web front-end.

– The second modification relates to the dynamic updating script[1] added to each output web page (i.e., a page that generate output for the end-users) to enable it to interact with the updater and perform automatic updating tasks of the page. The script generates a unique random number at the page's start-up and sends it to the updater in order to uniquely identify the opened instance of the page to the updater. At update times, the script disables HTML elements included in the page to make them not to accept any further requests and sends their states along with the generated id to the updater. After the update, the script retrieves the page's old state via the unique id passed through the new URL and loads it to the components. The mentioned script is usually injected into a few pages of the web front-ends, because the web front-ends (e.g., Joomla, SquirrelMail[2], and OCS) typically leverage *facade* design pattern [63] to emit output. For example, as seen in Section 7, only two pages of over 5,000 web pages generate output in Joomla.
– The third modification is to adjust the database connection string of the program by techniques similar to those described in the previous subsection.
– The last modification is to specify blocks of codes inside which the dynamic updates must not be applied due to the possibility of consistency violations. This is accomplished by incrementing a predefined counter at the beginning of the block and decrementing that counter at the end. If a block of code is not surrounded by this specification, ARESA automatically replaces the front-end irrespective of whether the code is executing or not.

Implementing ARESA, we faced five minor issues that will be discussed in the following where the approaches to direct them are also presented. The first issue is how to embed the dynamic updating script into the output pages. To this end, two methods can be employed: (1) an HTML frameset is used to encapsulate the original page in one frame and its dynamic updating code in another, or (2) an HTML script tag may be used to include the DSU code at the start of the original page. The former method is more modular while the latter is more efficient. In addition, the HTTP parameters sent to the frameset should be redirected to the original page in the former technique. A comparison of the above two methods is found in Section 7.

The second issue is on notifying web pages about events occurred on the server such as notifying them of the availability of updates. This can be accomplished using different techniques (e.g., Pushlets [64], Comet [65], Ajax [66], Jaxcent[3]). In ARESA, we use *long response* technique to achieve this effect. In this technique, every web page sends a *register* request to the updater at the start-up, and the server sends back a predefined character such as null character once a second as a response. When the updater stops sending the character, the web page interprets this as a new update having become available.

The third issue is about choosing a method through which independent threads of the updater know each other's state. For instance, the threads sending the null character to their registered pages should be informed about the other thread that receives the update notification form the ARESA runtime environment. To address this issue, the *file systems* approach has been used, in which a thread waiting for a special event checks the existence of a related file in the application's directory repeatedly. When an event occurs in a thread, it touches the related file in the application's directory and other interested threads are informed of that event.

The fourth issue is how to keep the list of opened pages held by the updater up-to-date. The reason for this is that if the list is not up-to-date, the updater may wait forever for a non-existent web page. To prevent this deadlock, the updater removes a web page from the list if it cannot send the null character to it, assuming the page has been unloaded.

The last issue is that the updater has to keep a list of the output pages in order to ask them to decline new incoming requests during the upgrade. To do so, a configuration file containing the required information is placed in the web front-end component of the enterprise patch and sent to the ARESA runtime engine at the redeployment times.

Fig. 3 shows an updatable web page along with the required DSU script and updater. Of note, we assume in this example that the page does not need to connect to database for brevity. Noting this, the page simply has been modified to include the script and a hidden HTML div element activated by the script at the update arrivals. The div's task is to inform the users about the update and ask them to wait awhile until the page is renewed. The page's original code follows with the exception of the programmer-specified blocks whose execution and the dynamic updating are mutually exclusive.

With respect to the space overhead, the updater has the least impact since there is a single instance of it added to the front-end. The second modification imposes a greater overhead on the size but it is also negligible, because it is added to only the output pages. As emphasized before, it is common amongst web programmers to create a few output web pages in which other configuration pages have been included. Existence

[1] The script has been implemented in Javascript, and therefore, can be incorporated into almost every web program.

[2] A web-mail client application, http://www.squirrelmail.org/

[3] http://www.jaxcent.com/

index.php:

```
<script ... src="index.js"/>
<div id="DSU" style="display:none">
  Upgrading the page...
</div>...
<?php
  $_SERVER['count']++;
  ...//The original main
  $_SERVER['count']--;
?>
```

updater:

```
function ready() {
  if($_SERVER['count']==0 && empty($pages))
    return true;
  else return false;
}
if($_GET['register']) {
  append($pages, $_GET['pageid']);
  while(!file_exists($update)){
    echo(chr(0));
    sleep(1);
  }
} else if($_GET['getstate']) {
  $lines=file($states);
  $line=find($lines,$_GET['pageId']);
  echo $line;
  unset($lines[$line_num]);
} else if($_GET['update_notif']) {
  touch($update);
  replace_output_pages();
  if(ready()) send_ready_to_aresa();
} else if($_GET['setstate']) {
  append($states,$_GET['pageid'],
    $_GET['state']);
  if(ready()) send_ready_to_aresa();
  wait_for_start_file();
  ... echo new_url();
} else if($_GET['start']) {
  unlink($update);
  touch($start);
}...
```

(a) Updatable web page, state transformer

index.js:

```
...
<!-- Utilities -->
function transformer(op,async,...) {
  xmlHttpReq.open(TRANS_ADDR+"?"+
    +op+"&pageId="+pageId, ...);
  ...
}
function waitForUpdate() {
  transformer("register", true,
    "updateCallBack");
}
function updateCallBack() {
  ...
  document.getElementById("DSU")
    .style.display="block";
  //Disable all HTML objects
  sendState();
}
function sendState() {
  ...
  <!-- Gather state -->
  transformer("setstate", true,
    "sendStateCallBack", state);
}
function sendStateCallBack() {
  window.top.location.replace(
    xmlhttpForState.responseText);
}
function getState() {
  transformer("getstate", false);
  ...<!-- Inject state -->
}
...
<!-- Extract pageId from URL -->
if(pageId==undefined) {
  pageId = Math.random()...;
} else {
  getState();
}
waitForUpdate();
...
```

(b) Javascript part of updatable web page

Figure 3: The source code of an updatable web application

of files like *config.inc.php* in most PHP applications is an evidence for this fact.

Regarding the time overhead, the added code to the output page along with its DSU script occupy more bandwidth loading them into the end-user's browser. In addition and similar to the desktop applications, getting the latest database connection string from ARESA runtime environment lengthens processing of dynamic web pages. Transmission of null characters between server and clients is another bandwidth overhead our approach incur. Of note, the null characters transmission also requires that the connection is opened during visit of the page. The updater, on the other hand, does not indeed incur overhead on the time or the bandwidth, because it resides in server and is not executed unless a request comes.

5.2 Patch application

The ARESA runtime environment is divided into three parts: (1) a *coordinator*, managing upgrade of front-ends

with their global database transformation, (2) a *Client Update Service* (CUS), responsible for updating a single desktop or web front-end and its local state, (3) and a *DataBase Update Service* (DBUS), updating the enterprise database. Fig. 4 illustrates these three parts and their relationships. In the next three subsections, we describe details and also approaches used in the coordinator, CUSs, and DBUS, respectively.

5.2.1 Coordinator

As described in Section 4, ARESA must strike a balance between preventing version inconsistencies as well as minimizing service disruptions based on whether the need for update of the global database. Coordinator is responsible for keeping this balance in ARESA. To support both cases, the coordinator goes through two distinct composite states: (1) 'software-and-db-updating' in which the coordinator obeys the inconsistency prevention rules, and (2) 'software-updating' in which the rules are relaxed in order to minimize service disruptions during redeployment.

Fig. 5 includes a UML statechart illustrating the coordinator's states and transitions amongst them. In both states, 'software-and-db-updating' and 'software-updating', the coordinator downloads the latest patches from the programmer's patch repository and notifies CUSs of the recent update instructing them to download the related component. After these steps and in the 'software-and-db-updating' state, the coordinator waits until all clients respond and then queries for interruption of their executions. The coordinator then steps into the 'waiting-for-pause-ack' sub-state and lingers until all acknowledge. Upon acknowledgment, the coordinator requests automatic patching of the database from DBUS and obtains the latest connection string from it. Finally, it asks CUSs to start the new programs.

In the 'software-updating' state on the other hand, the coordinator transitions to the 'sending-renew' sub-state immediately after sending the update notification message to the client applications and having received first acknowledgment from any one of them. In this state, the coordinator sends pause or start messages individually to a CUS that responds to the previous command irrespective of other clients' status. Contrary to our belief, this latter case was harder to implement, because the coordinator has less synchronization points to test the overall system and allowable statuses of each individual CUS. For example, it may be possible in the 'software-updating' state that a CUS starts the updated front-end while another is still downloading it.

Timeouts are used in the coordinator's states whenever a deadlock may occur in the system. For instance, in the 'waiting-for-ack' state in which the coordinator must wait for all clients to download the latest patches, it may transition to the next 'sending-pause' state regardless of any clients which have not acknowledged reaching a timeout. It is worth noting that clients which do not satisfy criteria determined in one state are excluded from the remaining redeployment process to avoid the aforementioned problems relating to consistency violations. In addition to the above tasks which are performed when an update is being applied, the coordinator has other duties fulfilled in its normal state. These include sending address of the latest version to the new clients or ones failed in applying the last update, registering CUSs of these new clients for future contacts, returning the latest database connection string to every legitimate requester, stating if a client is the latest version or not, etc.

It should be noted that the coordinator keeps the required information such as a list of updatable front-ends and the latest database connection string in a file which its contents is renewed accordingly when an automatic redeployment is performed or a new front-end is registered to the ARESA system.

5.2.2 DataBase update service

DBUS automatically updates the global database of the enterprise application when no program in the system uses that database. To do so, DBUS closes all opened connections to the database and instructs the DBMS not to accept any incoming connections while the patch is being applied. Specifically, it sets the maximum number of allowed connections to one. DBUS then executes a SQL script provided by the programmer and sent by the coordinator to patch the database. After db patching, DBUS sets the maximum number of allowed connections to the previous value. It also renames the old database in order to invalidate its connection string and no program can read from or write to that database. The DBUS finally gives the control back to the coordinator and sends also the newly generated connection string to it.

DBUS supports MySQL and Microsoft SQL server now. However, the aim has been to minimize changes to DBUS when other database management systems are incorporated into it. By contrast to the coordinator, DBUS does not require a configuration file in order to be deployed or executed.

5.2.3 Client update service

Client Update Service, or CUS for short, is responsible for updating a single application client in the system. The client can be either desktop or web-based. The major task of CUS is to transfer the state of the front-end from old to new version. As described in Section 4, this is performed by the programmer-specified state transformation approach.

At the update arrival, CUS first downloads the new version from the coordinator's file management system and places it in a specified location. The downloaded

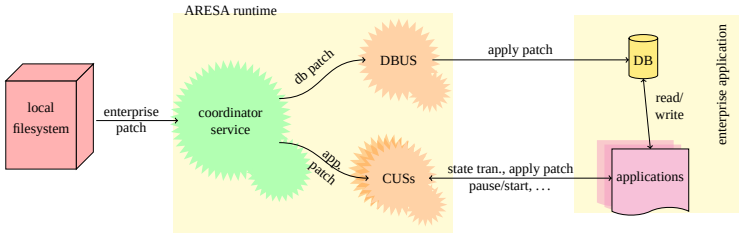


Figure 4: Patch application process in ARESA

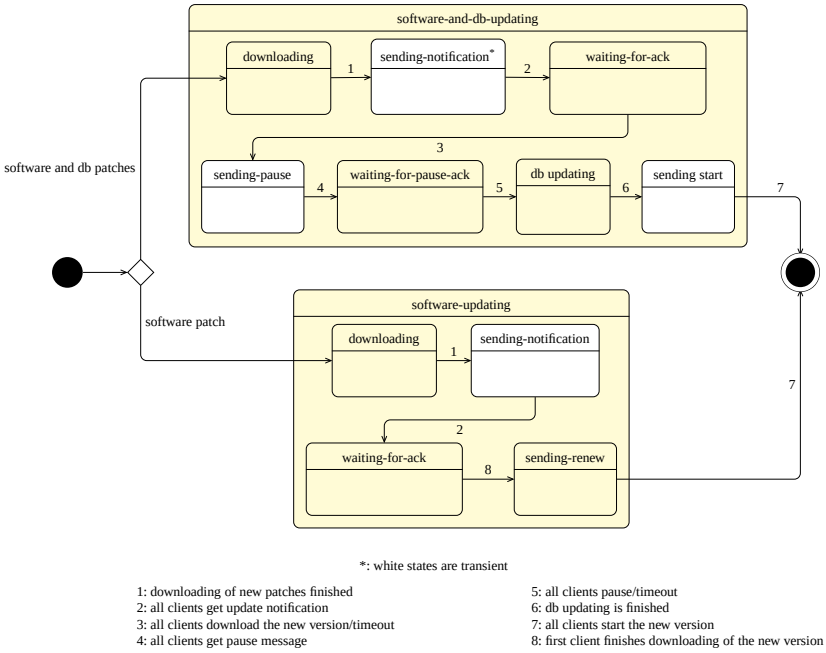


Figure 5: Coordinator's states illustrated by statechart

program is usually an executable file for the desktop front-ends (e.g., exe, jar) and a zip file containing pages for the web front-ends (e.g., html, jsp, php). The CUS then declares arrival of the update notification and lingers until the coordinator instructs to pause the front-end under its control.

Once the program receives the pause message and reaches the nearest update point, it sends the transformed state to CUS, asking it to store that state and then terminates. CUS in turn announces the interruption of the corresponding front-end to the coordinator and waits for its start message. When this message arrives, CUS starts the front-end's new version by executing a provided shell script. This script takes different approaches to start the desktop or the web front-ends. It creates an operating system process to run the former while it unzips the downloaded file in the application's root directory to renew the latter. Finally, CUS gives the stored state back to the application's new version upon request.

Other duties of CUS performed in the normal state include (1) asking the coordinator if its front-end is registered and updated, (2) requesting the front-end's latest version if is obsolete, (3) determining if an automatic redeployment is in progress, and (4) delivering the latest database connection string from the coordinator to the corresponding client application.

6 Formal proof

In this section, we aim to prove formally that the ARESA's update model and timing approaches do not violate systems' consistency. To this end, we first provide a formal definition of constraints that must be true in order for an enterprise application to be consistent, and then, we show formally that the mentioned constraints always hold true in the applications that employ ARESA for the automatic redeployments.

Definition 61*(Front-end $F_{i,k}$). Let $F_{i,k}$ denotes the version i of front-end F which uses the global database version k. F_i stands for a front-end F version i which may communicate with an unknown version of the global database. In addition, $F_{i,\times}$ denotes that the front-end F_i does not and also must not use any version of the global database.*

Definition 62*(Database D_k). D_k represents the version k of the global database. It is noted that there is only one version of the global database at a time in the system.*

Definition 63*(Dependency $\rightsquigarrow$). Let $\rightsquigarrow$ indicates dependency between two front-ends, say $F_i \rightsquigarrow G_j$ means that F_i depends on G_j. Dependency between two front-ends may be message passing or web service call.*

Definition 64*(Update $\mapsto$). Let $F_i \mapsto F_{i+1}$ denotes that the front-end F version i is updated to version $i+1$.*

Definition 65*(Consistency $\triangle$). An enterprise system with a set of front-ends and a global database is consistent at a specific time if and only if the following two conditions hold true:*

$$1.\forall F_i, G_j : F_i \rightsquigarrow G_j \Rightarrow i = j.$$
$$2.\exists! D_k \Rightarrow \forall F_{i,l} : (l = \times) \vee (l = k \wedge i \geq l).$$

In short, the former statement controls consistency among front-ends while the latter does so between front-ends and the enterprise database. The first statement places this constraint that the version of two dependent clients must be the same. For example, it should not be possible for a client of version 1 to send a message to another client of version 2. The reason for this is that the programs' semantics may be changed during upgrades. The second statement forces front-ends to either not use the database or use its latest version if they need to. For instance, if the latest version of the global database is 2, then front-ends requiring to access the global database must also use the database version 2. Of note, the notation $\exists! D_k$ means there is only one (and obviously the latest) version of database at any time in the system.

The other constraint imposed by this statement is that the versions of programs in the presentation tier should be higher than the version of database they use. For instance, versions of front-ends requiring to access the enterprise database must not be older than the version 2 in the above example. This ensures establishment of the aforementioned constraint on the system that if the enterprise database is updated, all front-ends must also be updated, although the converse might not be always true (the reason of this constraint is described in Sections 3 and 4).

Theorem 61*ARESA update model and redeployment timing do not cause consistency violations during and after redeployments.*

*Proof.*Here, we show that the two specified statements in the Definition 65 always hold true in ARESA. The first statement is valid all the time due to this consideration that enterprise application is data-centric in ARESA (refer to Section 3 for more information). In other words, there is no F_i and G_j in the application such that $i = j$ and $F_i \rightsquigarrow G_j$.

In order to prove the second condition in ARESA, we assume all possible states of the enterprise application and then check fulfillment of the condition in each state. Two coarse-grained states are assumed for the enterprise applications: (1) during the redeployment and (2) after the redeployment. In the former state, the application remains consistent because no front-end uses the database at that time (all db connections are closed and maximum number of connections is set to one). Specifically, for all $F_{i,l}$, $l = \times$ during applying the update patches.

The latter state, say after the redeployment, is divided into four distinct states based on whether the global database and the front-end have been successfully updated or not. First, (2a1) suppose that the database D_k has been updated and the old front-end $F_{i,l}$ converged to the specified update point in the proper time frame, meaning it also has been successfully updated. Formally speaking, $D_k \mapsto D_{k+1} \wedge F_{i,l} \mapsto F_{i+1,l+1}$. We must show $\triangle\{D_{k+1}, F_{i+1,l+1}\}$ in this case. Since the system has been consistent before redeployment, we have: $\triangle\{D_k, F_{i,l}\} \Rightarrow k = l \wedge i \geq l \Rightarrow k+1 = l+1 \wedge i+1 \geq l+1 \Rightarrow \triangle\{D_{k+1}, F_{i+1,l+1}\}$.

Second, (2a2) consider a state in which the database D_k has not been modified but the front-end $F_{i,l}$ has been paused timely and upgraded successfully. Therefore, we have $F_{i,l} \mapsto F_{i+1,l}$. According to this fact that the system has been consistent before the redeployment, we conclude: $\triangle\{D_k, F_{i,l}\} \Rightarrow k = l \wedge i \geq l \Rightarrow i+1 \geq l \Rightarrow \triangle\{D_k, F_{i+1,l}\}$.

Third, (2b1) we study a state in which the database D_k has been updated but the front-end $F_{i,l}$ has failed to update due to reaching its time-out. Specifically, what happed in the system is: $D_k \mapsto D_{k+1}$. Since ARESA renames the old database after upgrading it, no front-end including $F_{i,l}$ is able to use it after the update, meaning that l becomes $\times$ for this kind of front-end. Therefore, the system becomes a set of $\{D_{k+1}, F_{i,\times}\}$ which in turn is consistent according to the second statement of Definition 65.

In the fourth state, (2b2) the database D_k has not been changed and the front-end $F_{i,l}$ has failed to update because it has not converged to the update point in the predefined interval. Since the system has been consistent before the redeployment, we have $\triangle\{D_k, F_{i,l}\}$. On the other hand, neither the database nor the front-end has been modified during the redeployment, and therefore, the system remains $\{D_k, F_{i,l}\}$ after the redeployment which is consistent. The proof is now complete.

7 Experiment

Dynamic software updating systems devised to date have been evaluated by different types of benchmarks; from

desktop applications [4, 17, 33–35], to servers such as Apache, vsftpd[1], and PostgreSQL[2] [1, 2, 6, 18, 57]. Smith, et al. have classified the benchmarks used by the DSU systems in order to standardize the evaluation carried out in this area of research [67]. However, since these benchmarks focus on the application logic tier of the programs, they cannot evaluate the database or the web features of ARESA. Therefore, we have chosen three other benchmarks each of which can evaluate a specific part of ARESA. In this section, the experiences gained from applying ARESA to these benchmarks as well as the justification of choosing each one are described. When we performed our empirical studies, two evaluation metrics, time and space overheads, were more important to us, and therefore, values of only these two are shown for each study in the second subsection.

The first application, HABC (Hasib Activity Based Costing), is a cost calculator implemented in Microsoft C# 2008 and SQL server 2005. Six consecutive loops surrounded by two other nested loops calculate costs in HABC. Its database contains 90 tables and the front-end has been composed of 56 C# classes. In the upgrade chosen to be performed automatically in this paper, three database changes have occurred: (1) a table responsible for saving and then reporting some fine-grained cost information has been added, (2) in the product table, the type of user-entered code has been converted from *int* to *nchar* to support special characters such as '-' in the products' codes, and (3) in the users table, a column named 'active_year' has been added to enable each user to work with a desired financial year regardless of other users' active year.

HABC is distributed across two kinds of machines: (1) server which contains the database and one copy of it exists in target organizations, and (2) client that includes the HABC's executable and there are usually up to five copies of it deployed and running. Since HABC has been installed on several organizations in which multiple end-users work simultaneously, automatic and remote updating of it saves time. This along with dynamic updating of a C# application that accomplishes heavy calculations for the first time in the literature have encouraged us to choose HABC as one of our case studies.

Joomla, our second case study, is an open source content management system written in PHP and works mostly with MySQL. It has composed of 50 database tables and over 5,000 PHP files. Dynamically updating of Joomla, we evaluate web features of ARESA in a popular real-life environment. We chose to upgrade Joomla version 2.5.4 to 2.5.5 in order to update both the front-end and the database, and therefore, evaluate more parts of ARESA. In this test, the time overhead incurred by ARESA before and after the update has been measured.

[1] http://vsftpd.beasts.org/

[2] http://www.postgresql.org/

The last benchmark is a custom web application which has a single PHP and one MySQL table containing personal information of a community such as students of a class. The PHP page displays the information stored in the table through an HTML table tag. The upgrade patch renames column 'id' of the application's table to 'student_id' and also adds a column called 'address' to it. We measured the page's load time of both the conventional and the updatable versions of this application when different number of records are stored in the application's table. This helps us to find how different database loads affect the ARESA time overhead upon the web applications. In the following two subsections, the leveraged methods by which we have performed our tests and the results we obtained from them are described, respectively.

7.0.4 Method

In order to evaluate ARESA in automatic updating of HABC, we have used one physical and two virtual machines with the following specifications: (1) the first physical machine has a Core i3 CPU with the speed of 2.4 GHz and 4 gigabytes of RAM. Its operating system is Fedora 17 on which Java version 1.6.0_22 and Oracle weblogic 11g are installed. The coordinator has been deployed in this machine. (2) The second machine is installed on Virtualbox version 4.1.18 with one 2.4 GHz CPU and 1 gigabytes of RAM. Its operating system is Microsoft Windows 7 with SQL server 2005 and the HABC database installed on it. It also includes Oracle weblogic 11g on which the DBUS is installed and running. (3) The last virtual machine with the same hardware specification as the previous one executes Microsoft Windows 2003 server, Oracle weblogic 11g, and CUS deployed in it. This machine is responsible for running the HABC front-ends.

After running HABC, we have created a sample product with two production phases to be able to calculate the cost of at least one product. In addition, we interpolated a profiling code into the six parts of both the conventional and the updatable versions of HABC to compare the time consumed for the program's startup and loading five different pages of the application. We executed the conventional and the updatable versions of HABC alternatingly for a total of 10 cycles (i.e., 20 executions), measured the startup and pages loading times, and then, averaged all results to achieve more reliable numerals. In addition to the time, the space overhead of ARESA has been evaluated. The results are shown in the next subsection.

In order to make Joomla automatically updatable, one requires to know how the application generates results and where its database connection string is stored. Regarding the first question, we have found that only two PHP files, *index.php* and *administrator/index.php* generate the application's results. In other words, the

end-users invoke one of these pages and pass parameters of the information they looking for and the pages send back the generated results after including other pages. With respect to the second question, the database connection string is stored in a configuration file named *configuration.php* which is created at the setup time.

Therefore, making this application updatable merely requires injection of dynamic updating code into the index.php and administrator/index.php files and altering the updater to set the connection string of the patched database in the configuration.php file. As seen, Joomla's good design specification has minimized code modifications required by ARESA and consequently its overheads. We have made Joomla version 2.5.4 updatable by configuring DSUInjector to perform the above modifications to this version.

Similar to the HABC test scenario, we used three machines to evaluate ARESA's practicability and the time and size overheads it incurs in the Joomla upgrade. The first machine executing the coordinator is the same as its corresponding in the previous test case. The second and third machines are virtual, each of which has a single 2.4 GHz CPU, 1 gigabytes of RAM, and executes Fedora 16 as the operating system. The second computer includes MySQL server version 5.5, Oracle weblogic 11g, and a copy of DBUS installed and running. This machine plays the role of database server for the installation. The third computer includes Apache server 2.2, PHP version 5.3, Oracle weblogic 11g, and a copy of CUS installed on it. Both the conventional and the updatable versions of Joomla application are deployed in the Apache server of this computer.

After the installation has been completed, we used Apache JMeter[1] to measure and compare the response time of the conventional and the updatable front-ends to load their index.php pages in different users loads. To this end, we set JMeter to send 10, 50, 100, 150, and 200 requests to the both versions with the ramp-up time of 2 seconds for all of these requests. To be more accurate, we performed the tests alternatingly for a total of 10 times and then average the results, except for the test of 200 requests which has been executed three times. We have repeated the tests after both versions were updated to check if updates affect the performance of ARESA. The results are illustrated in the next subsection.

With regard to the last test scenario, we used the frameset technique described in Subsection 5.1 of Section 5 to make the custom web application updatable. Due to HTML framesets overhead, this results in evaluating the maximum degradation of performance and space ARESA may impose. In addition, the updatable index.php was configured to obtain the database connection string from CUS to cause the worst possible database connection time. The deployment structure and machines specifications in this scenario are the same as ones used for evaluating ARESA in the Joomla upgrade, and therefore, they are not described here.

In order to appraise the effect of different database loads on the performance degradation of ARESA, we requested both the updatable and non-updatable versions of the custom web application to display 10, 1000, 5000, 10000, and 20000 random records stored in their databases and then measured the response times of each request. Of note, since the index page of the updatable version is a frameset containing two other frames, we created three requests to be submitted to the updatable version instead of each request in the conventional one. Then, the sum of response times obtained from these executions has been considered as the response time of one request in the updatable version. These tests have been performed alternatingly and under the loads of 10, 100, 250, and 500 users. We used Apache JMeter to perform each test 10 times. Average time overhead results along with the space degradation of ARESA are illustrated in the next subsection.

7.0.5 Results

Table 1 shows the results of performance tests on HABC. As this table indicates, interpolating the dynamic updating features increases the startup time of HABC from 5.6 to 6.9 seconds. On the other hand, the form loading and calculation times remain nearly unchanged. One time the updatable code took longer and once again the conventional program took more time to execute. Based on this information, we conclude that ARESA increases the time overhead of this desktop application by 19% and only at the startup time.

We observed in the evaluations performed on Joomla that the time complexity functions of both the updatable and the conventional versions grow similarly, either before or after the upgrade. The maximum difference happened after the update with the 200 users load in which the average response time of the conventional Joomla was 1498.1 seconds while the corresponding time in the updatable version was 1750.5 seconds. This means that the average time overhead incurred by ARESA on real-life web applications is 15%. Fig. 6 illustrates the evaluation results performed on Joomla in two charts. Functions of dashed and solid lines show the response times of the conventional and the updatable front-ends, respectively.

The most significant time overhead imposed by ARESA occurred in the custom web application at high database loads. As Fig. 7 indicates, this happened when 250 or 500 users simultaneously request the program's main page to fetch and display 5000 records from the database. In the first case (250 users), the response times of the updatable and the non-updatable versions were 85.157 and 48.699 seconds, respectively. This means that ARESA increases the response time of the page by 43% in this circumstance. In the second case (500 users), the

[1] http://jmeter.apache.org

Table 1: Performance comparison of updatable and conventional executables of HABC

Startup elapsed times:

sys.	1	2	3	4	5	6	7	8	9	10	average
upd.	6279	7050	6949	6892	6399	6819	6209	10737	6168	6279	**6978.1**
norm.	6299	5628	5197	5598	5537	5387	5808	6349	5507	5527	**5683.7**

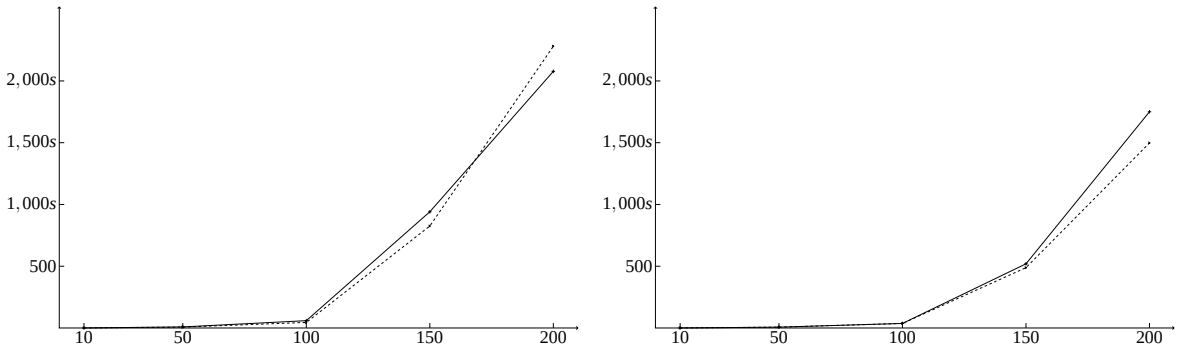
Page loading elapsed time in updatable HABC:

no.	material	equivalent units	cost allocation	cost	calculating cost
1	400[1]	230	200	150	520
2	360	240	210	190	751
3	370	270	210	160	490
4	721	490	280	250	630
5	781	370	340	210	690
average	**526**	**320**	**248**	**192**	**616**

Page loading elapsed time in conventional HABC:

no.	material	equivalent units	cost allocation	cost	calculating cost
1	600	250	250	230	530
2	400	240	200	200	460
3	390	270	230	180	450
4	440	230	180	210	530
5	660	300	450	220	580
average	**498**	**258**	**262**	**208**	**510**

[1]Times are in milliseconds.



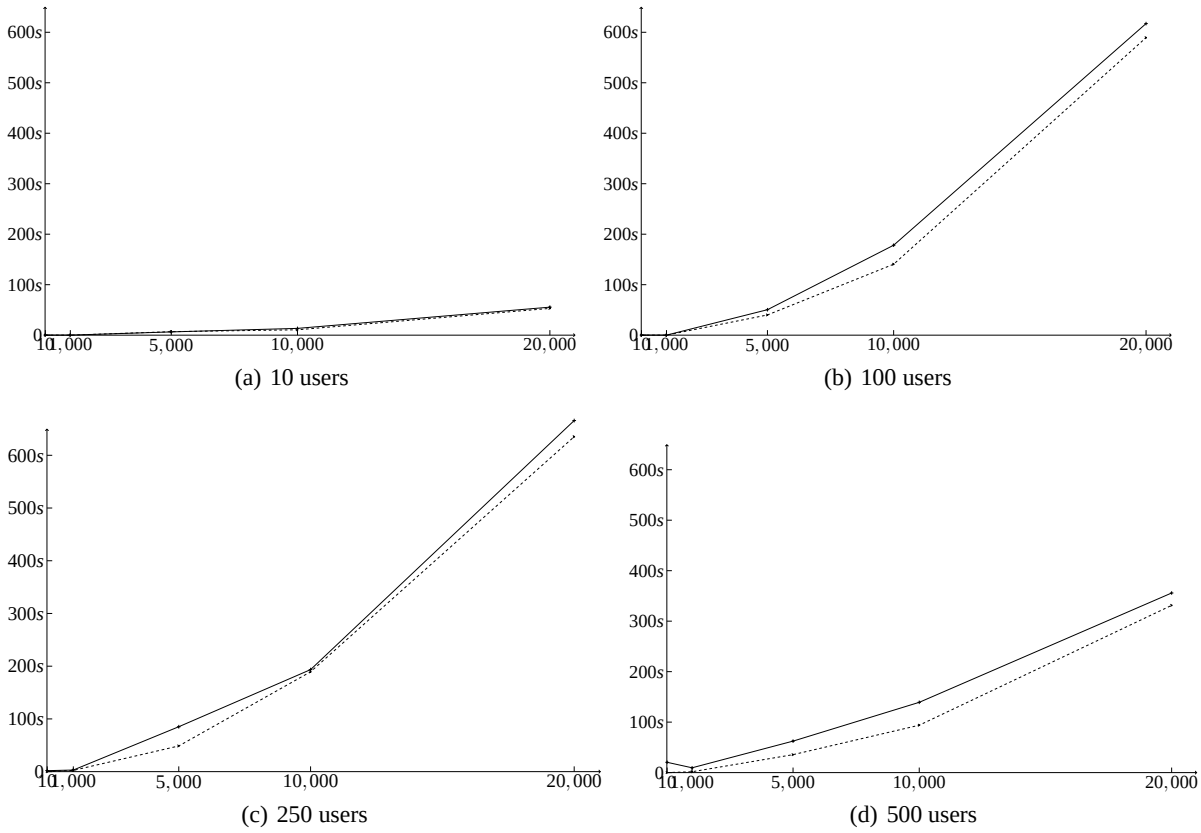
(a) before redeployment (b) after redeployment

Figure 6: Performance comparison of conventional and updatable versions of Joomla

response times of the updatable and the non-updatable pages were 62.366 and 35.549 seconds, respectively. As in the first case scenario, ARESA degrades performance by 43%.

It is however noted that since the custom web application is very small and also we used inefficient dynamic updating techniques in this scenario, ARESA had significance impact on the program's size and performance. A real-life application ought to spend a lot of time in business logic layer, and therefore, the impact of ARESA significantly decreases as the size and complexity of the application approaches those of real-life applications. As another point, the response time of the custom web application was decreased when the number of users grew from 250 to 500. The reason is that an average of 20 percent of requests failed in the 500 users workload.

We measured and summarized sizes of both the updatable and non-updatable test cases in Table 2. This table indicates that ARESA somewhat imposes the space overhead on the commercial and real-life applications less than 1%. However, this overhead increased to more than 3 times for the custom web application which contains only one web page made updatable by the space-occupying frameset technique.



(a) 10 users (b) 100 users

(c) 250 users (d) 500 users

Figure 7: Performance comparison of conventional and updatable versions of custom web application

Table 2: Size comparison of updatable and conventional applications

system	HABC		Joomla		custom web	
	conventional	updatable	conventional	updatable	conventional	updatable
source	32,292,536[1]	32,299,211	18,920,034	18,931,621	4,737	13,645
binary	9,962,496	9,965,568	N/A	N/A	N/A	N/A

[1]Sizes are in bytes.

8 Conclusion and future work

This paper presented a framework, called ARESA, to automatically redeploy data-centric enterprise systems. ARESA provides a novel update model as well as a unique update timing technique in order to preserve the consistency of applications undergoing the automatic redeployments while the end-users just experience short and predictable disruptions. Practicability and correctness of this approach is proved both formally and empirically. We used ARESA to dynamically update programs written in web and desktop programming languages, some reported for the first time in the software updating literature. The experimental results suggest ARESA provides at most 19% performance penalty in dynamically updating of applications with real-life complexities. We plan to extend ARESA in the future, particularly in the following important areas:

–*Omitting the data-centric constraint*: ARESA automatically redeploys enterprise applications in which front-ends communicate with each other through a central database. We are planning to extend ARESA to support a wider range of applications.

–*DB comparison*: two further features in DBComparator can make automatic redeployment of enterprise applications simpler. First, DBComparator must support renaming and moving of database elements such as tables and columns. Second, it should generate SQL patches in addition to only report the differences. We are working on adding these functionalities in addition to supporting other database management systems.

–*Dynamic Updating injection*: Currently, DSUInjector requires the programmer's intervention in two stages: (1) determining how a new database connection string should be set in the updated web and desktop front-ends, and (2) specifying which web files dynamic updating code should be injected to. We plan to omit the human intervention required in these two stages of the dynamic updating injection.

–*Dealing with timeouts*: defining appropriate timeout after which automatic redeployments are performed irrespective of the running programs has proved to be a challenge in ARESA. Short timeouts could make more front-ends hitting the limit, and therefore, be excluded from the rest of the redeployment process. Long timeouts may cause some front-ends to wait unnecessarily for other processes to terminate before an update can be applied. We plan to perform an empirical study determining appropriate timeouts for several categories of software applications.

References

[1] M W Hicks, J T Moore, and S Nettles. Dynamic software updating. In *SIGPLAN Not.*, pages 13–23, New York, NY, USA, 2001. University of Pennsylvania, ACM.

[2] I G Neamtiu. *Practical dynamic software updating*. PhD thesis, University of Maryland, College Park, 2008.

[3] Andrew Baumann, Jonathan Appavoo, Robert W Wisniewski, Dilma Da Silva, Orran Krieger, and G Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[4] Allan Raundahl Gregersen, Michael Rasmussen, and BoNø rregaard Jø rgensen. State of the Art of Dynamic Software Updating in Java. In José Cordeiro and Marten van Sinderen, editors, *Software Technologies*, volume 457 of *Communications in Computer and Information Science*, pages 99–113. Springer Berlin Heidelberg, 2014.

[5] D.-Y. Lin and I Neamtiu. Collateral evolution of applications and databases. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol '09, pages 31–40, New York, NY, USA, 2009. ACM.

[6] S Subramanian, M Hicks, and K S McKinley. Dynamic software updates: A VM-centric approach. *SIGPLAN Not.*, 44(6):1–12, 2009.

[7] Pamela Bhattacharya and I Neamtiu. Dynamic updates for web and cloud applications. In *Proceedings of the 2010 Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications*, APLWACA '10, pages 21–25, New York, NY, USA, 2010. ACM.

[8] M Wahler, S Richter, S Kumar, and M Oriol. Non-disruptive Large-scale Component Updates for Real-Time Controllers. In *Proceedings of the 3rd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '11, Hannover - Germany, April 2011. IEEE Computer Society.

[9] C A Curino, H J Moon, and C Zaniolo. Graceful database schema evolution: The PRISM workbench. *Proc. VLDB Endow.*, 1(1):761–772, August 2008.

[10] S Guo, H Li, C Ding, and H Ren. Study on Large-Scale Embedded Databases Evolution. In W Du, editor, *Informatics and Management Science II*, volume 205 of *Lecture Notes in Electrical Engineering*, pages 153–158. Springer London, 2013.

[11] Mario Pukall, C Kästner, W Cazzola, S Götz, Alexander Grebhahn, R Schröter, Gunter Saake, K Christian, G Sebastian, and Reimar Schr. JavAdaptorFlexible runtime updates of Java applications. *Software: Practice and Experience*, 43(2):153–185, 2013.

[12] Tudor Dumitras, P Narasimhan, and Eli Tilevich. To upgrade or not to upgrade: Impact of online upgrades across multiple administrative domains. *SIGPLAN Not.*, 45(10):865–876, October 2010.

[13] H Seifzadeh, H Abolhassani, and M S Moshkenani. A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568, 2012.

[14] Microsoft. How To Deploy an ASP Application to Another Server by Using Internet Information Server, 2014.

[15] Oracle. Enterprise Manager Lifecycle Management Administrator's Guide, 2014.

[16] B Noyes. *Smart Client Deployment with ClickOnce: Deploying Windows Forms Applications with ClickOnce*. Pearson Education, 2006.

[17] S Malabarba, R Pandey, J Gragg, E Barr, and J F Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, pages 337–361, London, UK, 2000. Springer-Verlag.

[18] H Chen, J Yu, R Chen, B Zang, and P.-C. Yew. POLUS: A POwerful Live Updating System. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society.

[19] Y Vandewoude. *Dynamically updating component-oriented systems*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering, K.U.Leuven, Leuven, Belgium, 2007.

[20] R S Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 470–476, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[21] Insup Lee. *Dymos: A dynamic modification system*. PhD thesis, The University of Wisconsin - Madison, 1983.

[22] Multicians. MultiCS Dynamic Linkage Features, 2009.

[23] J Appavoo, K Hui, C A N Soules, R W Wisniewski, D M D Silva, O Krieger, M A Auslander, D J Edelsohn, B Gamsa, G R Ganger, P McKenney, M Ostrowski, B Rosenburg, M Stumm, and J Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Syst. J.*, 42(1):60–76, 2003.

[24] S Potter and J Nieh. AutoPod: Unscheduled System Updates with Zero Data Loss. In *Second International Conference on Autonomic Computing*, pages 367–368, 2005.

[25] A Baumann. *Dynamic update for operating systems*. PhD thesis, Computer Science & Engineering, Faculty of Engineering, UNSW, 2007.

[26] K Makris and K D Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 327–340, New York, NY, USA, 2007. ACM.

[27] J Arnold and M F Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 187–198, New York, NY, USA, 2009. ACM.

[28] J Montgomery. A Model for Updating Real-Time Applications. *Real-Time Syst.*, 27(2):169–189, 2004.

[29] G Gracioli and A A Fröhlich. An operating system infrastructure for remote code update in deeply embedded systems. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, HotSWUp '08, pages 31–35, New York, NY, USA, 2008. ACM.

[30] Habib Seifzadeh, Ali Asghar Pourhaji Kazem, Mehdi Kargahi, and Ali Movaghar. A Method for Dynamic Software Updating in Real-Time Systems. In *Proceedings of the 2009 Eigth IEEE/ACIS International Conference on Computer and Information Science*, ICIS '09, pages 34–38, Washington, DC, USA, 2009. IEEE Computer Society.

[31] A C Noubissi, J Iguchi-Cartigny, and J.-L. Lanet. Hot Updates for Java Based Smart Cards. In *Proceedings of the 3rd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '11, Hannover - Germany, April 2011. IEEE Computer Society.

[32] M Dmitriev. *Safe Class and Data Evolution in Large and Long-Lived Java Applications*. PhD thesis, Department of Computing Science, University of Glasgow, 2001.

[33] A Orso, A Rao, and M Harrold. A Technique for Dynamic Updating of Java Software. In *Software Maintenance, IEEE International Conference on*, page 649, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[34] R P Bialek. *Dynamic Updates of Existing Java Applications*. PhD thesis, Faculty of Science, University of Copenhagen, 2006.

[35] G Bierman, M Parkinson, and J Noble. UpgradeJ: Incremental Typechecking for Class Upgrades. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 235–259, Berlin, Heidelberg, 2008. Springer-Verlag.

[36] Deepak Gupta and Pankaj Jalote. On line software version change using state transfer between processes. *Softw. Pract. Exper.*, 23(9):949–964, September 1993.

[37] G Altekar, I Bagrak, P Burstein, and A Schultz. OPUS: Online patches and updates for security. In *Proceedings of the 14th conference on USENIX Security Symposium*, SSYM'05, page 19, Berkeley, CA, USA, 2005. USENIX Association.

[38] K Makris and R A Bazzi. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *Proceedings of 2009 USENIX Annual Technical Conference*, 2009.

[39] Luís Pina and Michael Hicks. Rubah: Efficient, General-purpose Dynamic Software Updating for Java. In *Presented as part of the 5th Workshop on Hot Topics in Software Upgrades*, San Jose, CA, 2013. USENIX.

[40] Christopher M Hayden, Edward K Smith, Michail Denchev, Michael Hicks, and Jeffrey S Foster. Kitsune: Efficient, General-purpose Dynamic Software Updating for C. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 249–264, New York, NY, USA, 2012. ACM.

[41] S Ajmani, B Liskov, and L Shrira. Modular Software Upgrades for Distributed Systems. In *Object-Oriented Programming*, volume 4067 of *LNCS ECOOP 2006*, pages 452–476, 2006.

[42] S van der Burg, E Dolstra, and M de Jonge. Atomic upgrading of distributed systems. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, HotSWUp '08, pages 1–5, New York, NY, USA, 2008. ACM.

[43] V P La Manna. Dynamic software update for component-based distributed systems. In *Proceedings of the 16th international workshop on Component-oriented programming*, WCOP '11, pages 1–8, New York, NY, USA, 2011. ACM.

[44] C Boyapati, B Liskov, L Shrira, C.-H. Moh, and S Richman. Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, 38(11):403–417, 2003.

[45] B Liskov, A Adya, M Castro, S Ghemawat, R Gruber, U Maheshwari, A C Myers, M Day, and L Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 318–329, New York, NY, USA, 1996. ACM.

[46] Cristiano Giuffrida. *Safe and Automatic Live Update*. phdthesis, VU University Amsterdam, 2014.

[47] G Hjálmtýsson and Robert Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, page 6, Berkeley, CA, USA, 1998. USENIX Association.

[48] M Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*, 2001.

[49] Habib Seifzadeh, Mostafa Kermani, and Mohsen Sadighi. Dynamic Maintenance of Software Systems at Runtime. In *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, pages 859–865, Washington, DC, USA, March 2008. IEEE Computer Society.

[50] M Jalili, S Parsa, and H Seifzadeh. A Hybrid Model in Dynamic Software Updating for C. In D. Ślkezak and T.-h. Kim and A. Kiumi and T. Jiang and J. Verner and S. Abrahão, editor, *Advances in Software Engineering*, volume 59 of *Communications in Computer and Information Science*, pages 151–159. Springer Berlin Heidelberg, 2009.

[51] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Trans. Softw. Eng.*, 33(12):856–868, 2007.

[52] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. *SIGPLAN Not.*, 44(6):13–24, 2009.

[53] ITIC. Database Competition Heats Up. http://itic-corp.com/, 2010.

[54] D Gupta, P Jalote, and G Barua. A Formal Framework for On-line Software Version Change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996.
[55] G Bierman, M Hicks, P Sewell, and G Stoyle. Formalizing Dynamic Software Updating. In *On-line Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE)*, 2003.
[56] K Makris. *Whole-program dynamic software updating*. PhD thesis, Arizona State University, 2009.
[57] J Stanek, S Kothari, T N Nguyen, and C Cruz-Neira. Online Software Maintenance for Mission-Critical Systems. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 93–103, Washington, DC, USA, 2006. IEEE Computer Society.
[58] C Giuffrida and A S Tanenbaum. Cooperative update: A new model for dependable live update. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 1–6, New York, NY, USA, 2009. ACM.
[59] M Hashimoto. A Method of Safety Analysis for Runtime Code Update. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, pages 60–74, 2007.
[60] J Buisson and F Dagnat. ReCaml: Execution state as the cornerstone of reconfigurations. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 27–38, New York, NY, USA, September 2010. ACM.
[61] J Buisson and F Dagnat. Introspecting continuations in order to update active code. In *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, pages 1–5, New York, NY, USA, 2008. ACM.
[62] Kishore Channabasavaiah, Kerrie Holley, and Edward Tuggle. Migrating to a service-oriented architecture. *IBM DeveloperWorks*, 16, 2003.
[63] E Gamma, R Helm, R E Johnson, and J Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
[64] Just Van Den Broecke. Pushlets: Send events from Servlets to DHTML client browsers. *JavaWorld*, 2000.
[65] P McCarthy and D Crane. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Apress, New York, NY, USA, 2008.
[66] J J Garrett. Ajax: A new approach to web applications. http://www.adaptivepath.com/, 2005.
[67] E K Smith, M Hicks, and J S Foster. Towards Standardized Benchmarks for Dynamic Software Updating Systems. In *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, HotSWUp '12, Zurich - Switzerland, June 2012. IEEE Computer Society.

Habib Seifzadeh received his Ph.D. in Software Engineering from Islamic Azad University, Science and Research Branch in 2013. His areas of research include software engineering, programming languages, and algorithms. Besides over ten years of academic experience, Habib also has extensive practical experience gained by being involved in several big and small projects on enterprise applications. He is now an assistant professor with the computer engineering faculty of Islamic Azad University, Najafabad Branch.

Hassan Abolhassani received his Ph.D. from Saitama University of Japan with a thesis on Automatic Software Design focusing on Learning from Human Designers. His areas of academic research include software automation, semantic Web researches, knowledge-based software design, and design patterns. He worked as Senior Technologist providing software based solutions for top-level clients in Japan when he was with Xist-Interactive (Razorfish Japan), until the end of September 2004, when he joined Sharif University of Technology as an assistant professor. He is now an associate professor with the computer engineering department of Sharif University of Technology.

Mohsen Sadighi Moshkenani received his B.S. in Mathematics and Statistics from Shiraz University, Shiraz, Iran, in 1973, and his M.S. in Computer Engineering from Sharif University of Technology, Tehran, Iran, in 1977, and his Ph.D. in Computer Engineering from Indian Institute of Science (IISc) Bangalore, India, in 1991. He has over three decades of professional experience in well known universities of Iran: Shahid Beheshti University, Isfahan University of Technology, and Sharif University of Technology-International Campus at Kish Island. His research interests are knowledge engineering, semantic Web, software engineering and education. Dr. Sadighi Moshkenani is a member of ACM and Informatics Society of Iran.