Theseus: Whole Updates of Java Server Applications

Erwann Wernli
Software Composition Group
University of Bern

***Abstract*—We present a novel approach to update server applications in Java. In our approach, different versions of the code coexist in the system, but are isolated into distinct *contexts*. The server can switch from one context to another incrementally in order to process incoming requests. Our approach has the following characteristics: (1) updatability is defined and added to the application by developers, (2) no syntax is added to the language and the update is controlled via the manipulation of objects, (3) the whole system is migrated lazily and eventually replaces the old system, (4) it is safe to update global entities anytime and there is no need to reach update points. We demonstrate our approach by updating the Jetty web server.**

I. Introduction

Many techniques to update systems dynamically, *i.e.*, without a restart, have been devised for C, C++, and Java applications. Three categories of dynamic updating systems exist: systems that install updates immediately, systems that install updates when no methods impacted by the update are on the stack, and systems which install updates at specific update points. The first category can lead to run-time type errors and executions of incoherent mixes of old and new code. The second category ensures type-safety, but incoherent mixes of old and new code can still lead to an incorrect application behavior. The third category constrains the installation at specific points that are known to be safe. It provides the strongest safety but requires also the most efforts. The second and third categories imply timing constraints that can be hard to meet and can compromise the timely installation of updates in multi-threaded systems [1], [2].

We propose Theseus[1], an update scheme for session-based server applications (web servers, telnet servers, message brokers, etc.). In our approach, different versions of the code coexist in the system, but are isolated within distinct *contexts*. Contexts avoid the problem of incoherent mixes of old and new code, and do not impose timing constraints. The server can switch from one context to another incrementally in order to process incoming requests. This is, in spirit, similar to a traditional switch between two instances of an application running in two distinct virtual machines. However, our aim is to replace the application *within the same virtual machine*. To maintain the overall consistency of the system, objects that are shared between different versions have two distinct versions as well. We keep them consistent with one other thanks to bidirectional transformations. This strategy alleviates the need to reach update points in order to update global entities. Our approach is reflective and the switch between the versions is controlled manually by the developer: contexts are first-class entities that can be manipulated at run-time.

We describe in this paper our update model and present an implementation of our approach in Java. We demonstrate our approach by updating the Jetty web server, for which we did not measure a perceptible overhead. We also show that for idiomatic server code, our approach does not entail many changes.

[1]In reference to Theseus' paradox.

II. Theseus in a Nutshell

The core abstraction of our approach is the *context*. Code executes within a context, and a given object can have a different structure, or a different behavior depending on the current context. Contexts are first-class, and the current context can be queried anytime. Contexts resolve class names to actual classes definitions, which define the behavior of existing objects. Contexts enable context switches. This is an inherent dynamic operation whose signature is the following: `Context.invoke(Object receiver, String method, Class[] signature, Object[] params)`. It resembles the traditional reflection API. However, instead of executing the method in the current context, it executes the operation in the given context.

To accommodate structural changes to objects, contexts form a list and each context in the list encodes a pair of *transformation functions*. One function defines the transformation of objects from the old to the new context, and another one the transformation from the new to the old context. There exists at startup the `Root` context. Contexts are regular Java classes, which must implement the `Context` interface. When a transformation is executed, it receives two distinct versions of the *state* of a given object. Transformations must manipulate the states reflectively.

Figure 1 illustrates the use of contexts. A thread executes the `CommandReader` runnable that reads commands from the input stream and processes them. A `CommandReader` has state `currentPath` which is used and updated by the various commands. A new command is introduced in the code of version 2 (lines 10-12). After the processing of each command, the application checks if a certain file

978-1-4673-1764-1/12/$31.00 © 2012 IEEE

```
1.   public class CommandReader implements Runnable {
2.      String currentPath = ".";
3.      public void run() {
4.         while( true ) {
5.           String[] line = readInput();
6.           if( line[0].equals("cd")) {
7.              currentPath = line[1];
8.           } else if ( line[0].equals("ls")) {
9.              printDirectory( currentPath );
10. *        } else if ( line[0].equals("pwd")) {
11. *           printPath( currentPath );
12. *        }
13.
14.          if( ! fileExists( "patch") ) {
15.            continue;
16.          }
17.          // Signal file is present, so we update
18.          String contextClass =
19.              readAndDeleteFile( "patch" );
20.          Context newContext =
21.              currentContext().newSuccessor( contextClass );
22.          newContext.invoke( this, "spawn", null, null );
23.          return;
24.        }
25.      }
26.      public void spawn() {
27.         new Thread(this).start();
28.      }
29.
30.      // rest of the class (printDirectory, etc.)
31.
32.  }
```

Figure 1. A sample self-updatable application. The application processes commands from the input stream. Updates are signaled via the presence of a file. Lines 10-12 marked with a star indicate the changes between the two versions.

named `patch` is present (line 14). If this is the case, the file contains the name of the new context class that must be loaded. The new context is loaded and instantiated reflectively via `currentContext().newSuccessor( String className )`. The application then updates itself by invoking method `spawn` on itself (the receiver of the invocation in line 22 is `this`), but in the newest context. It creates and starts a new thread which executes the newest code, before the old thread terminates with `return`. The update does not imply structural changes and the state is transferred during the update with the default identity transformation. The `currentPath` is not lost. Contexts can encode transformations, if necessary. Eventually, when no threads are referencing the old context, it can be garbage collected as can be the corresponding versions of the state. This example illustrates the core characteristics of Theseus:

- *User-defined updatability.* Applications must be manually adapted to become "updatable". The way updates are triggered is not hard-coded.
- *First-class contexts.* Contexts are objects. No special syntax is added and updates are realized solely via the manipulation of objects, possibly reflectively.
- *Whole update.* Old and new code coexist in the system, but contexts prevent incoherent mixes of old and new code at the thread level. Eventually, the whole system is replaced and runs in the newest version.
- *No update points.* Code can be executed in a context immediately, even if the context changes the structure of global entities in the system.

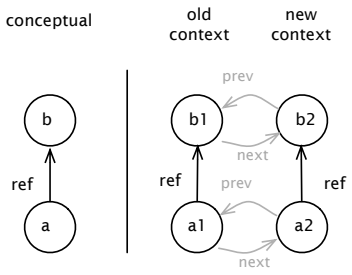

Figure 2. Conceptually, field `ref` of object `a` references object `b`. Implementation-wise, we have objects `a1`, `a2`, and `b1`, `b2` that reference each other. Information in gray is not visible to the application.

III. Implementation

Conceptually, any object has as many states as contexts. In practice, one is interested to migrate only the state that is shared between contexts. The migration happens lazily to accommodate large heaps. Once an object has been migrated, it has two versions of its state. To keep these two versions coherent, the system runs bidirectional transformations in a lazy fashion: an update in one version only invalidates the other one, which will only be updated if it is accessed again.

We assume a correctly synchronized program, *i.e.*, without concurrent reads and writes to a given field. Consequently, we do not synchronize the bidirectional transformations. This implies that bidirectional transformations must update fields individually (not the whole object), in order to ensure that concurrent reads and writes to independent fields remain thread-safe.

Before the old context can be garbage collected, the system must ensure that all reachable objects have been migrated, and force their migration if necessary. This task is conducted during the finalization of the context, and does not block the progress of the rest of the system. Such forced migration might conflict with migrations from the application, which implies that migrations of objects must be exclusive.

In previous work, we have implemented our model as a program transformation in Smalltalk [3]. Each context is a distinct type universe, and objects that have been migrated have two versions (of distinct types) that reference each other. Figure 2 displays such a settings. Conceptually, field `ref` of object `a` references object `b`. Implementation-wise, we have objects `a1`, `a2`, `b1` and `b2`.

The program transformation intercepts and rewrites the accesses to state. When a field is accessed, the system checks whether the value is outdated. If this is the case, the system updates the field using information in the other context. Let us say for instance that `a2` is outdated, and that field `a2.ref` contains `null` instead of a reference to `b2`. When `a2.ref` is accessed, the system notices it is outdated. To update the reference, the system locates the older version `a1` of `a`, navigates the reference `ref` to `b1`, and obtains the newest version `b2`.

Java Specificities: We describe here how such a program transformation has been achieved for Java, and highlight some of the difficulties related to this target language.

1) *Non-updatable classes.* Certain classes of the system are non-updatable, *i.e.*, they are passed as-is between contexts. Only very core classes are treated as such: `java.lang.*`, `java.io.*`, `java.net.*`. This was necessary because they use native methods that are not bound any longer if the class is instrumented. The classes in `java.util.*` are updatable, except locks from `java.util.concurrent.locks.*`.
2) `Object` *class.* The `Object` class cannot be extended with the necessary information for our model (namely the outdated flag, the references to the previous and next versions). Instead, we extended the subclasses of `Object`, as long as they were updatable.
3) *Class loader.* Contexts are special class loaders that enable class names to be rebound to different concrete classes in different contexts.
4) *Class vs. instance.* Classes and instances cannot be treated homogeneously since we need to distinguish between accesses to instance fields and static fields. Different versions of a class reference each others using static fields. Static initializers must be bypassed when loading the newest versions of existing classes.
5) *Primitive types.* Primitive types are not contextualized, and can be copied across contexts.
6) *Constructor.* Objects can be instantiated only using an appropriate constructor, even reflectively. To ensure that the system is able to instantiate classes without a "no-arg" constructor, we add a synthetic constructor with a special signature.
7) *Nested classes.* The bytecode of the synthetic constructor generated by the compiler is peculiar and is Javassist unfriendly. We excluded from the instrumentation accesses to `this$` and `val$` in the synthetic constructor.
8) *Arrays.* Arrays with primitive types can be copied as-is across contexts. If an array contains objects, the system must keep two versions of the array if it is shared. Unfortunately, there is no way to extend the underlying array class with the necessary information for our model (see item 2). We keep this information in a weak hash map, which implies one level of indirection.
9) *Synchronization.* Concurrency control provided by the language via `synchronized`, `wait` and `notify` will not work for objects with several versions. We keep a "contextual" identity that is shared between different versions of an object, and these instructions must be adapted to use the contextual identity, *e.g.*,`synchronize(this.id){ ... }` instead of `synchronize(this){ ... }`. Synchronization with first-class objects from `java.util.concurrent.locks.*` works as is.
10) *Garbage collection.* A "handle" of the context is used to keep track of who is using the context. Threads hold a reference to the handle of the context they were initially created in. Reflective invocations with `Context.invoke(...)` also obtain a handle. When no handles are referenced any longer, the context can be garbage collected.
11) *Native methods.* Native methods could directly manipulate the state of objects that would escape our interception mechanism. In practice, this never happens as native methods are few, and operate usually on primitive types, or types that are non-updatable.

Our implementation is available on github at https://github.com/ewernli/theseus. We use Javassist and ASM to instrument the code.

Table I
EVOLUTION OF JETTY.

version	classes impacted	add.	rem.	mod.
6.1.4	-	-	-	-
6.1.5	11	9	3	5
6.1.6	4	6	0	0
6.1.7	3	3	0	2

Table II
LOAD TESTING OF JETTY.

version	average	min	max	throughput
original	6	3	15	1077
instrumented	6	3	18	1090

IV. EVALUATION

A. Updating Jetty

We applied our approach to update the Jetty web server. We adapted the class `AbstractConnector.Acceptor` which accepts incoming connections in a way similar to Figure 1. We performed updates between versions 6.1.4, 6.1.5, 6.1.6, 6.1.7 (but not in one sequence). Table I shows the number of fields that were added (add.), removed (rem.), or modified (mod.), and the number of classes that were impacted (classes). A modification can be a renaming, or

a type change. Fields that were added could be initialized with default values.

Web servers are IO-bound, so we conducted a load test to assess the overhead of our approach instead of measuring times on the server side. The load test used 10 threads, each performing 300 requests to serve various files and directories. Jetty was installed on a server (8 Xeon 2.33 Ghz CPUs) and JMeter on a workstation (Intel Core 2 Duo 3 Ghz). We assume the effects of the network traffic to be constant. Table II shows the time per request and the throughput for the original and instrumented versions. We did not notice a perceptible overhead. We conclude that our implementation can be used to update real-world applications like Jetty.

B. Idioms in Thread Management and Infinite Loops

Our approach implies that the target application must be adapted to become updatable. Context-related code will typically be added in long-running loops (lines 17-23 in Figure 1). In the example, restarting the thread is very easy (see lines 26-29). However, if such a loop is deep in the call stack, the situation might be more complicated. In essence, what is needed is to capture a continuation, update it to match the new context, and call it. In our Smalltalk implementation, closures and stack frames were first-class and could be passed across contexts.

Java has no such support for continuations, and the simulation of continuations requires invasive changes in the sources [4]. Also, server applications usually depend on higher level abstractions to control threading (*e.g.*, thread pools) and do not favor direct creation and scheduling of threads (line 27 in Figure 1). To assess the effort needed to adapt server applications to our approach we investigated a few Java applications:

- *Tomcat.* To our surprise, Tomcat implements by default a pattern that fits our approach. Method `LeaderFollowerWorkerThread.runIt` accepts an incoming connection, and schedules itself for another execution in another thread using a thread pool.
- *Jetty.* Jetty implements a runnable whose method `Acceptor.run` uses a traditional `while` loop to accept incoming connections.
- *JDNSS.* The method `Protos.run` uses a `while` loop that creates and starts threads directly.
- *JES.* The method `Pop3Processor.run` accepts connections in a `while` loop and handles them directly.

From this (brief) study, we conclude that no context-related code will require invasive changes as methods that need to be adapted are usually in runnables that can be passed to thread pools easily. Thread pools must however be configured or adapted to eventually "renew" the whole pool in order for the update to eventually complete. For our experiment, we made the classes implementing the pool non-updatable and added 8 lines of code so that the pool obtains a "handle" (see section III) when a job is accepted, and releases the handle when the job has been executed.

V. Discussion

In our implementation, all state accesses are intercepted to check whether the state in that particular context is outdated, in which case the state must be updated. Most objects are local to a context though, and the (unique) state of such objects is never outdated. The check entails nevertheless an overhead. It is hard to benchmark the overhead, as it greatly depends on the data structures in use, and the access patterns. Our evaluation shows that in practice this overhead has only a marginal impact on the overall performance.

Two directions can be explored further to reduce this overhead: (1) add and remove the check on demand using the HotSpot method redefinition facilities. (2) instrument only classes whose instances are shared across contexts.

Before the old context is garbage collected, the system forces the migration of all objects reachable from the new context. How effectively the old representations can be reclaimed during this process depends on the structure of the object graph, but we can expect objects to be reclaimed incrementally. We leave the study of our approach for applications with large heaps for future work. How contexts could be supported at the virtual machine level in order to improve performance is also deferred to future work.

VI. Related Work

Techniques to dynamically update software have been implemented with program transformations, custom virtual machines, or modified runtimes libraries. The implementation we described is a program transformation. This work builds upon previous work in Smalltalk [3].

The most recent approaches for Java are Javadaptor [5], Javaleon [6], DCEVM [7], JVolve [2]. The two first techniques in the list [5], [6] are based on program transformations. The main problem in this case is to deal with the so-called "version barrier": new and old versions of a type are not compatible. To solve this incompatibility, Javadaptor instruments caller with an extra "container" field to which any type can be assigned. It rewrites getters and setters to use this container instead of the original fields. Javaleon uses another technique coined "in-place proxification". Instead of a wrapper that replaces the original object, the original object is modified to forward the call to its newest version. To forward the call, parameters and return values must be adapted to match the corresponding versions. Forwarding relies on heuristics and mapping old and new behaviors might not be possible. We do not forward invocations and do not face such problems, but similarly to Javaleon we convert objects between type universes. In our approach and in Javaleon, field accesses are intercepted in order to migrate the state lazily. Our instrumentation results in minimal changes (only a check is added to getter and setters)

and does not rely on a level of indirection. If the check is removed when the update is completely installed (see section V), the code running after the update is identical to the original. Unlike Javaleon and Javadaptor, we use contexts to prevent incoherent mixes of old and new code.

The two last techniques in the list [7], [2] rely on modified VMs. JVolves redefines classes when no methods on the stack are impacted. For increased safety, the user can manually specify additional methods that must not be active. We do not face similar constraints. The DCEVM allows old and new code to coexist, and updates can be installed any time the program can be suspended. This offers essentially the same flexibility as with dynamic languages, at the price that certain operations might fail, yet gracefully (a runtime error is raised).

The most recent approaches for C/C++ are Ginseng [1], UpStare [8], Kitsune [9], and POLUS [10]. Ginseng, UpStare and Kitsune require threads to reach update points (manually defined or possibly statically computed). This is unlike POLUS which alleviates this constraint by using bidirectional transformations. Duggan had previously proposed to use bidirectional object converters in order to avoid runtime type errors [11]. We use bidirectional converters like POLUS and Duggan, but we control the coexistence of old and new code with more discipline via the notion of contexts. Like Kitsune we update the whole program, and developers must adapt the program so that the execution can be resumed at the desired location (see subsection IV-B). UpState uses stack reconstruction in order to do so without requiring manual changes to the code.

VII. Conclusions

We have proposed a novel approach for dynamic updates of server applications in Java, and demonstrated it by updating the Jetty web server. Our conclusions are the following:

- Evolutions of programs during maintenance entail small changes to the fields of classes. Changes are mostly additions or removals, and the unmodified fields map one to one between versions. A bidirectional transformation is easy to achieve in most cases.
- The notion of contexts is convenient to use and does not bloat the code. For idiomatic server applications, only small changes should be required in order to make them updatable.
- The overhead of our instrumentation does not entail visible degradation of the performance for non-memory intensive applications.

We plan to mature our implementation and apply it to other server applications in order to assess further its applicability. We notably plan to study the characteristics of memory intensive applications.

Acknowledgment

We would like to thank Jorge Ressia and Oscar Nierstrasz for reviews of earlier drafts of our paper. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 - Sept. 2012).

References

[1] I. Neamtiu and M. Hicks, "Safe and timely updates to multi-threaded programs," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 13–24.

[2] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: a VM-centric approach," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 1–12.

[3] E. Wernli, M. Lungu, and O. Nierstrasz, "Incremental dynamic updates with first-class contexts," in *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2012*, 2012, to appear.

[4] L. Stadler, C. Wimmer, T. Würthinger, H. Mössenböck, and J. Rose, "Lazy continuations for java virtual machines," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '09. New York, NY, USA: ACM, 2009, pp. 143–152.

[5] M. Pukall, A. Grebhahn, R. Schröter, C. Kästner, W. Cazzola, and S. Götz, "Javadaptor: unrestricted dynamic software updates for Java," in *Proceeding of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 989–991.

[6] A. R. Gregersen and B. N. Jørgensen, "Dynamic update of Java applications — balancing change flexibility vs programming transparency," *J. Softw. Maint. Evol.*, vol. 21, pp. 81–112, mar 2009.

[7] T. Würthinger, C. Wimmer, and L. Stadler, "Unrestricted and safe dynamic code evolution for Java," *Science of Computer Programming*, Jul. 2011.

[8] K. Makris and R. A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, ser. USENIX'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 31–31.

[9] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for C," 2012.

[10] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew, "Dynamic software updating using a relaxed consistency model," *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 679–694, 2011.

[11] D. Duggan, "Type-based hot swapping of running modules," in *Intl. Conf. on Functional Programming*, 2001, pp. 62–73.