Abstract

The requirement for 24/7 availability of distributed applications complicates their maintenance and evolution as shutting down such applications to perform updates may not be an acceptable solution. Therefore, there is a need to be able to update these applications dynamically, i.e. without shutting them down. Current solutions for building dynamically updatable Java applications require that applications either are prepared for updates from the outset, comply with a specific framework, or are executed in a modified virtual machine. In this work, we present a novel approach to creating dynamically updatable Java applications based on the concept of partitioning applications into units of dynamic updates and illustrate how this approach better addresses the problems of adding update support to existing applications than traditional approaches.

1. Introduction

The growing popularity of the Internet as well as the emergence of supporting tools and frameworks for building distributed applications leads to more and more applications being distributed and inter-connected. Nodes of such inter-connected applications require other nodes to be continuously available to function correctly. On the other hand, running applications need to be maintained and/or adapted to new requirements. Unfortunately, distribution and the requirement for 24/7 availability of running applications complicate their evolution as shutting down these applications to perform updates might not be acceptable in all situations. Therefore, there is a need to be able to update these applications at run-time.

To address the problem of dynamically updating Java applications, a number of frameworks that support dynamic evolution have been proposed. The Java Management Extension (JMX) [11], for example, is an architecture that supports replacing manageable objects (so-called MBeans) at runtime. This framework is used by Java application servers such as JBoss to transparently support replacing of JavaBeans at runtime [6]. However, this framework only allows for dynamic update of JavaBeans. Other frameworks such as SOFA [12] and Formaware [14] support dynamic evolution of Java applications by introducing the notion of updatable components. In these frameworks, updatable applications need to be built using the updatable components concept.

Many existing applications, however, are not built using JavaBeans, SOFA, Formaware, or any other framework that offers support for dynamic updates. To allow dynamic updates of such applications, other solutions have to be found. One approach to solve this problem is to modify the Java Virtual Machine (JVM), as presented in [4] and [5]. In both these approaches, a level of indirection on method calls is introduced so that calls can be redirected at run-time, allowing for dynamically replacing classes. Unfortunately, using a modified virtual machine is not an acceptable solution in all situations as it would require the modified JVM to run on all nodes of a distributed application and, therefore, disables the use of a standard JVM. Furthermore, unloading of a class at run-time and replacing it by a modified version may cause a considerable performance overhead.

In this work, we propose an approach where Java applications can be made dynamically updatable without either modifying the JVM or forcing developers to use an updatable framework. Inspired by the J-Orchestra project [17] and the Addistant translator [16] that present solutions to transparently partition Java applications to achieve distribution, we propose to (i) group the classes of a Java application into partitions and (ii) add functionality to dynamically update entire partitions. The partitions of an application encapsulate the classes they contain and have explicit provided and required interfaces, making them similar to components [15]. Whereas the information about the partitioning can be either provided by a developer or automatically extracted from source code, supporting dynamic updates of partitions can be enabled by using an appropriate framework, e.g., DUCS [1].
tion 2, we further discuss our motivation to add support for dynamic updates to Java applications, followed by an illustration of our partitioning approach in Section 3. In section 4, we present how partitions facilitate dynamic update of existing applications and evaluate benefits and drawbacks our approach in Section 5. We conclude this paper with a discussion of related work in Section 6 and outline our contribution and present suggestions for future work in Section 7.

2. Background

The goal of our work is to enable on-the-fly updating of object-oriented applications with a special focus on applications written in Java. In this section, we present background information on (i) considered applications, i.e. we briefly present what assumptions we make on the types of applications considered for dynamic update, (ii) dynamic updates of classes, i.e. we give an overview about issues arising when supporting dynamic updates of classes (which is the basis of our approach of updating partitions), and (iii) motivation for partitioning, i.e. we illustrate our motivation to use partitions as units of updates.

2.1. Considered Applications

Throughout the rest of this work, we make the following assumptions about applications that we consider candidates for dynamic update:

- The applications are correct and robust, i.e. they compile without error messages and run without crashes for extensive periods of time.
- They are provided as byte code in class files (no source code is required).
- They execute on a standard Java Virtual Machine (no custom JVM is required).
- They use any of the standard class loaders to load classes at run-time.

In the following sections, when referring to an application’s code, we refer to the byte code of the application and not to its source code.

2.2. Dynamic Updates of Classes

The basic operation behind the update of any Java application is the replacement of the byte code of a class (or a set of classes) by its new version. No matter what technique is used to update the corresponding byte code, several problems are introduced that need to be addressed in any updatable system. These problems include compatibility, object identity, class unloading, state transfer, and performance overhead. They are briefly discussed in the following.

Compatibility: dynamic updates of applications must not compromise correctness and stability of the updated applications. Correctness refers to both type safety and semantical correctness. To guarantee type safety, updatable frameworks often allow the update of an implementation of a class, but not its interface, or allow incremental interfaces changes only [6, 14]. To guarantee semantical correctness, approaches based behavioral protocols can be used [13].

Identity of objects and classes: when we dynamically replace the byte code of a class by new byte code, we also must make sure that all instances of this class are updated accordingly. Furthermore, all program entities that refer either to this class or any of its instances have to be updated as well, i.e. they now refer to the new version(s) and not the old one(s). To support correct object identification in such a situation, an indirection mechanism may need to be provided [3].

Class unloading: to support dynamic updates, we need to be able to unload obsolete versions of classes. Current JVM’s support class unloading in a very limited way, i.e. by unloading of a ClassLoader. Classes loaded by a different class loader can not refer to one other unless they are loaded by the same parent class loader. This may lead to the creation of a hierarchy of class loaders, which is not desirable for long living applications whose structure changes over time. Furthermore, there is the possibility that cyclic class dependencies are introduced.

State transfer: as mentioned above, the update of a class requires a corresponding update of all of its instances. This implies that we have to transfer the state of an object from its old version to the new version. This does not only apply to all instance variables, but also to local variables of methods as well as the current stack counter, if an object is currently active. Hence, a safe update point [7] must be indicated (by either a developer or a code analysis tool) in order to perform state transfers in a correct way.

Performance: the support for dynamic updates requires some level of indirection so that method calls are directed to the correct objects and classes, respectively. As this level of indirection is required for every updatable class, some run-time performance degradation will occur for all updatable classes.

In Section 4, we illustrate how these problems are addressed by our partitioning approach.

2.3. Motivation for Partitioning

Using single classes as units of update is a sound way to support dynamic updates, but has a number of drawbacks:

- It enforces a fine-grained approach, which is not always necessary, e.g., when updating whole libraries.
Every updatable class must be checked if its update is compatible with its previous version.

The class update support introduces overhead on every call to an updatable class.

An application may consist of a number of complex components (that include many classes) and they usually do not need support for a (fine-grained) update of single classes.

Instead of enforcing update granularity on single classes, we propose to group classes into **updatable partitions**. Based on the size of the partitions (i.e. the number of classes they contain), we can control the amount of overhead introduced by the support for dynamic updates. Additionally, partitioning provides a way to preserve modularity (programming module separations) at run-time which is generally lost when an application is started. Preserving modularity can ease the process of updating applications as changes to program modules can be incorporated into the running application.

### 3. Partitioning Approach

Our approach to achieve support for partitioning of applications is as follows:

1. After an application is fully developed it can be partitioned using the provided **partitioning specification**. The specification includes information about how many partitions to create and what classes each partition includes. For details refer to section 3.2.

2. A transformation tool uses the partitioning specification to modify an application’s byte code by replacing direct cross partition method invocations with **remote invocations**. For details refer to section 3.4.

3. During the transformation, information about partition dependencies is extracted from the code and made accessible to support future updates. For details refer to section 3.5.

Throughout the rest of this work, we will use the sample Java application given in Figure 1 in order to illustrate our approach of partitioning and issues introduced by partitioning.

### 3.1. Basic Concepts

Our partitioning model uses the concepts of **partitions** (or partition specifications) and **partition mediators** which will be illustrated in the following.

```java
import java.util.*;

public class A {
    private B local;
    private C remote=
    public A() {
        local=new B();
        local.incr();
    }
    public void initC(D d) { remote=new C(d); }
    public String m(D caller) {
        D rem=remote.getD();
        if(caller==rem) return "Equal";
        return "Different";
    }
}

public class B {
    private int number=0;
    private void incr() { number+=2; }
    protected String hello() {
        incr();
        return "B"+number;
    }
}

public class C {
    private D rem;
    public C(D d) { rem=d; }
    public D getD() {
        return rem;
    }
}

public class D extends B {
    private A internalA;
    public D() {
        super();
        internalA = new A();
        internalA.initC(this);
    }
    public void run() {
        String result;
        do {
            hello();
            result=internalA.m(this);
        } while(result.equals("Equal"));
    }
}
```

Figure 1: Example Java application

**Partitions**: a partition is a static concept describing the grouping of the classes of an application. For a particular partition, we distinguish two kinds of classes: **local classes** (i.e. classes which belong to the same partition) and **remote classes** (i.e. classes belonging to other partitions). Consequently, local and remote method invocations refer to invocations on local and remote objects, respectively. Whilst method invocations on instances of local classes are performed directly, method calls to objects of remote classes are performed indirectly: they go through a separator, which we call a **partition mediator**. Please note that a class can be a member of multiple partitions or not belong to a partition at all.

**Partition Mediators**: a partition mediator (mediator in short) is a runtime abstraction representing a wrapper for classes and their instances in a given partition and serves as a proxy
for remote invocations. A mediator separates classes and their instances that perform calls across partitions and provides the required transparency for updating classes and objects. A mediator manages classes and instances of one partition only.

In Figure 2, an illustration of a partitioning of our sample Java application is given: it contains three partitions (Part1, Part2, and Part3) as well as a partition mediator for each of the three partitions. Partition Part1 contains the classes A and B, partition Part2 the class C and partition Part3 the class D.

Figure 2: A model of a running application grouped into three partitions.

3.2. Providing Partitioning Specification

One of the requirements of our approach is that we have a partitioning specification available for an application that requires dynamic update support. Such specifications can either be defined by an application developer or extracted from source code by an analysis tool. A description of how to obtain such a specification for a given application is beyond the scope of this work. It is however important to note that the choice of partitioning may have a considerable impact on the performance of dynamic updates as well as on the flexibility what kind of updates can be performed.

In Figure 3, we illustrate an example of a partitioning specification for the application given in Figure 1. Each partition has a name (needed to support its identification when updates occur) as well as a list associated classes and/or package names. In the latter case, all classes belonging to that package will be assigned to the corresponding partition. Classes not included in the specification (such as, for example, java.lang.String used the classes A, B, and D) are assigned to the partition(s) that they are used in.

3.3. Issues Introduced by Partitioning

Although our approach provides a way to split up an application into updatable partitions, it introduces a number of problems that either originate from the problems related to dynamic class updates (refer to Section 2.2) or are directly related to our partitioning approach.

Inheritance across partitions. When parent and child classes are separated, a child object instance requires the parent object to be instantiated in its partition. We use the approach of J-Orchestra [17] that utilizes delegating proxies to support cross-partition inheritance. This approach has the advantage that an update of a (parent-)class is directly reflected in all its derived classes. We present details of the approach in section 3.4. A disadvantage of this approach (apart from introduced overhead) is that cross partition delegations require the parent class to be instantiated before the child class, which is not possible if the parent class is abstract. This difficulty can be solved by providing implementations of abstract methods to delegate calls to their child classes.

Overlapping partitions. Our model allows for classes belonging to more than one partition. This can become necessary to meet specific performance requirements (local method invocations do not suffer from any run-time overhead). If a class is included in more than one partition, it can evolve independently in each partition. When replacing such a partition, all instances of a class in that partition are replaced, while instances of this class in other partitions are not affected. Having the same classes with different versions in different partitions is not problematic because potential incompatibilities between classes are handled by partition mediators (otherwise such update can not be performed). If one needs to have the same version of a class in all partitions, then all partitions need to be replaced.

Static methods and fields. In order to correctly access static methods and/or fields of a class, we additionally need to introduce class proxies. Class proxies and object proxies share the same functionality and they can both be represented by partition mediators.

Figure 3: Partitioning specification

<partition name=Part1>
<class>A</class>
<class>B</class>
</partition>
<partition name=Part2>
<class>C</class>
</partition>
<partition name=Part3>
<class>D</class>
</partition>
3.4. Transforming an Application

Upon loading of each class, our modified class loader calls a transformer that alters the loaded class. The transformer uses the byte code manipulation tool Javassist [2] to modify the byte code at load time. Every additional class loaded by the modified class loader is recursively altered by the transformer as well.

To support class replacement, classes are being renamed and a link from the original class name to a unique name is kept for each class. When replacing a class, the original name stays the same but the renamed class name changes. Access to the original name is included for every updatable class through a reflective interface MetaIF, illustrated in Figure 4. The transformer automatically generates MetaIF’s methods for each updatable class and adds an implement MetaIF to all updatable classes.

```java
interface MetaIF {
    MetaIF getMetaIF(String attrName);
    Object getValue(String attrName);
    Object invoke(String methodName, Object[] args);
    String getClassID(); // return real class id
    String getClassID(); // return object reference
}
```

Figure 4: Meta interface

Using MetaIF, mediators can invoke methods as well as read and alter attributes of the updatable objects. For static methods and fields, a static version of the method `invoke` as well as static `get` and `set` methods are generated.

After renaming all updatable classes and adding MetaIF and its associated methods, the transformer alters the following constructs in the corresponding byte code:

- `new`: in Java, the operator `new` is used to instantiate objects of a given class. If `new` is used on a remote class, it is substituted by the operator `remotenew` that represents the code responsible for contacting the remote mediator and invoking an instantiate operation on the remote mediator. The returned reference is a remote reference. In Figure 1, lines 8 and 34 are places where `new` is replaced by `remotenew`.
- `super`: when a class uses a `super` call to access behaviour in its parent class and its parent is in a different partition, the call is replaced with a delegate operation. In constructors, calls to `super` are replaced by `OID=delegateNew(className, args)` that represents the code responsible for (i) contacting the mediator of the parent class, (ii) creating an instance of the parent class, and (iii) returning the object identifier (OID) of this newly created parent class instance. This OID is assigned to the newly created object because the object ID of both the parent and child class instances have to be the same.

For further information about this delegation approach, refer to [10]. On line 33 of Figure 1, the `super()` call is replaced as illustrated above.

- `instanceof and this operators`: when comparing objects with remote objects or checking if an object is an instance of a remote class, we need to use the functionality of the meta interface illustrated in Figure 5 to access the real class and object ID, respectively. For example, `obj==this` is replaced by `obj.getOID().equals(this.getOID())`. Such a transformation is applied at line 11 in Figure 1.

Similarly, `o instanceof B` is replaced by `o.getClassID().equals("B")`.

- `Remote references`: method invocations on remote objects and/or remote classes are replaced by calls to the local mediator: `call(remoteRef, method, args)` represents the code responsible for (i) calling the embracing mediator for potential adaptations (refer to Section 4), (ii) contacting the remote mediator, (iii) invoking the called method, and (iv) adapting the result of the invocation. In the example in Figure 1, we use remote invocation on lines 10, 35, and 41.

- `Remote attributes`: attributes of remote classes and/or objects cannot be accessed directly, but have to be replaced by a method call. More specifically, the code for reading and writing the attribute `foo` of remote object `obj` is replaced by `obj.getAttribute("foo")` and `obj.setAttribute("foo", aValue)`, respectively.

3.5. Building Partition Dependencies Information

To check whether or not an update of a partition is compatible with the rest of the application, we need to address the compatibility issue presented in Section 2.2. An updated partition is compatible with the rest of the application if its dependencies are satisfied. These dependencies consist of all remote invocations and remote instantiations (invocations of constructors) that were substituted in the transformation process.

Remote invocations going from one partition to another define a set of required interfaces. Remote invocations coming from other partitions to the partition define a set of provided interfaces. Hence, both provided and required interfaces consist of cross-partition calls only and do not include any local calls.

During the partitioning transformation, the provided and required interfaces are build for every partition. Upon an update of a partition, its interfaces are regenerated. An example of a partition’s interface is presented in Figure 5. The example shows that partition `Part1` provides two constructors and three methods to other partitions. The information about required interfaces is used to check updates of other partitions (that their provided interfaces satisfy the required inter-
4. Support for Dynamic Updating

Mediators can be considered as the equivalents of ConfMgr introduced in the DUCS framework [1]. They provide an update interface to receive update requests that include new code, potential adapters, state transfer functions, and mapping descriptors (if class names change).

Before a partition can be updated, its mediator checks if the new implementation’s provided and required interfaces do not break dependencies to other partitions. The interface of a partition provided by an update request is generated using the same transformation tool as when partitioning the application. The provided interface of the new partition’s implementation must at least include the provided interfaces of the previous version. If the validation fails, the update is rejected, otherwise the partition can be replaced.

To replace a partition dynamically, the mediator starts a new update process in a thread that is executed in parallel to the application process. The update process utilizes mediator’s dynamic data structure and support for object’s state transfer. We first describe the dynamic data structure and support for state transfer, and then we present the update process.

4.1. Mediator’s Data Structure

Mediators are responsible for supervising partition replacements. In Figure 6, we illustrate a mediator’s internal data structure which enables dynamic updates.

For each object being created using the remotenew operation, an entry is added to the instance table of the enclosing partition. The instance table includes the unique OID of each object, a reference to this object as well as some status information. The status field is used to indicate if the referenced object is in the process of executing requests. The status field can be extended with statistical data like average, minimum, or maximum execution time of invocations. The statistical data can support the update process when waiting for an executing call to finish before starting an update process.

Figure 6 illustrates the implementation of Part1’s mediator. The mediator includes three objects and one proxy: two instances of the class B where the class name was renamed to B1 to allow substitutions, and one instance of the class A. Instances referred to from the internal instance table can be referred to from the outside. Object A1 refers to one local object (it was created using new operator within the class A) and one proxy object to access C (it was created using remotenew). In the status field of the instance table we see that the instance of the class A with OID=2 is running (its status data is 1). The status is increased when entering a method and decreased when the call is finished.

4.2. State Transfer Function

One of problems introduced by dynamic updates is the transfer of an object’s state (refer to Section 2.2). Similarly, this problem also applies to partitions. Transferring a partition’s state consists of transferring states of all objects included in this partition using a State Transfer Function (STF). The STF for a partition must include mappings of states of old partition’s objects to their corresponding new versions. Such a mapping is necessary for a continuous and correct service of that partition after an update. For the scope of this paper, we assume that all STF’s are correct, i.e. that state is correctly transferred between versions.

To effectively transfer states of objects, a mapping from old to new class names needs to be provided, otherwise we assume that class names are unchanged.

The STF for a partition is expressed in the constructors for the partition’s new classes. Consider the STF for a new implementation of the class A.

```java
public A(MetaIF old) { // STF
    par1 = new B(old.getMetaIF("local"));
    par2 = old.getValue("remote");
}
```

The constructor receives a reference to its “old” object. Using this reference, we can read the old object’s attributes using their identifiers. We can also invoke the STF on other
components by calling their constructors. For example, at line 2 we first read the value of the attribute \texttt{local}, the result is then passed to the STF constructor of the new implementation of the class \texttt{B}.

### 4.3. Update Process

The update process executes in a separate thread for each partition of the application allowing the application to continue its execution. Updating a partition’s mediator, however, blocks new requests to the updating partition to guarantee correct updates.

Because the blocking of calls can lead to a deadlock situation, the mediator maintains a call history to allow calls originating in the same partition to be executed.

Only after all internal executions have terminated, the update process can start. For every externally accessible object, the mediator runs the STF using the STF constructor. When finished, all the old externally accessible objects have been updated to their newly created versions. Then, the update process enables access to the partition and the application can continue its normal execution using the new versions of the updated partition’s objects.

### 5. Discussion

**Solutions to the inherited dynamic update problems.** In the partitioning transformation (presented in section 3.4), we have addressed the problems introduced by supporting dynamic updating. The \textit{Compatibility} problem no longer relates to single classes – the compatibility of updates is now checked on the level of partition’s provided and required interfaces. The \textit{Identity of object and classes} is supported by extending classes with additional information about class and object ID, which is kept persistent during updates. \textit{Class unloading} is supported by renaming classes within partitions. \textit{State Transfer} is supported by partitions state transfer functions (discussed in Section 4.2).

**Significance of the introduced overhead.** Depending on the granularity of the partitions, the introduced overhead can vary. On the one hand, we want to have small partitions, which are easily updatable, and on the other hand, we do not want considerable runtime overhead. We believe that this conflict can be solved by “wisely” choosing partitions.

**Remote procedure calls.** Distributed applications are naturally separated by the remote procedure calls layer. Therefore, nodes of a distributed application can be regarded as partitions. To support updates of inter-connected distributed nodes, a provided/required interface needs to be added to the application stubs and proxies, something we will address in future work.

**Stabilized classes.** Stabilized classes are classes that are not included in the partitioning specification and hence cannot be updated (e.g., system classes). On the other hand, such classes may keep references to updatable classes, e.g., the class \texttt{java.lang.Vector}. Because such classes are not intended to be updated, we assume that they do not change and are local to a partition and can be accessed directly. However, the classes they refer to may be updated at runtime. As a consequence, stabilized classes must use the mediator to access classes local to the same partition.

### 6. Related Work

Our research is related to the areas of \textit{application partitioning, process migration}, and \textit{updatable component architectures}.

J-Orchestra [17] and Addistant [16] provide a way to divide an application into partitions. The aim of these projects is to transparently support distribution to ease the process of building distributed applications.

In [7], an approach on utilizing state transfer functions (STF) to support seamless updates of running applications is presented. Hicks et al. [8] address the issue of automatically generating STF to support on-the-fly updates of applications written in C. This approach uses source code analysis to generate the STFs, an idea that can be extended to Java Byte Code.

Our previous research addressed an architecture of an updatable application called DUCS [1]. DUCS is a similar approach to SOFA [12] which uses component managers (in this paper referred to as mediators) to control components. FORMAware [14] is an other approach to support dynamic updates of component architectures. FORMAware proposes to enforce a link between the architectural description and the update process, as well as utilizing composite components to control coordinated updates. We use this approach in DUCS by involving architecture managers in managing the updates. In the future, we plan to link update operations with the creation of architecture description.

Dynamic C++ Classes [9] provide an approach utilizing proxies to support dynamic updates. We therefore believe that our partitioning approach can be generalized to other languages than Java.

### 7. Conclusions and Future Work

We have presented a novel approach to divide Java application into smaller, manageable partitions that can be updated dynamically. This work is a continuation of our research on building dynamically updatable systems [1]. The presented approach has the following benefits:

- Partitioning can be applied to existing Java applications executing in standard JVMs.
Partitioning is transparent and it may be performed after an application has been fully developed.

Partitioning eases the dynamic update process as it supports reasoning of what is being updated and what effect it has on the rest of an application.

Using partitions as units of updates is more flexible than previous approaches supporting class updates as partitions can include single classes or whole program packages.

Generated partitions can be regarded as updatable components creating a link between legacy and component based applications.

Partitioning supports dynamic evolution of applications as it provides a way to reason about what is being updated, what is being affected by an update, and simplify how the updates can be performed. Partitioning is performed by reading a partition specification that defines how an application is to be divided. The specification may be provided after an application is developed so partitioning can be freely adjusted to meet the needs of developers after a finished development cycle.

Partitioning introduces flexibility on the cost of performance. Our approach provides a tool for developers to decide whether their application shall be faster but inflexible (static) or slower but dynamic. We think that having a freedom to choose the applications characteristics after finishing the updates, at deployment time or at run-time, is an important tool supporting evolution.

In future work, we plan to continue our research in defining guidelines for optimal partitions with respect to flexibility vs. performance. We will evaluate different partitioning configurations and their effect on easiness of updates and performance of the application.

To further support dynamic updates of applications, we recognize the need to repartition existing partitions. Repartitioning covers operations on partitions such as merging, dividing and moving classes across partitions. Repartitioning may be necessary due to changing design, rearranging classes or performance optimizations.

Additionally, we plan to extend partitioning to use remote procedure calls as partition boundaries, include support for stabilized classes and for in-process updates of partitions.

References


