

# DOSGi: An Architecture for Instant Replication

Jörg Domaschka, Holger Schmidt, Franz J. Hauck  
*Institute of Distributed Systems*  
*Ulm University*  
{joerg.domaschka,holger.schmidt,franz.hauck}@uni-ulm.de

Rüdiger Kapitza  
*Informatik 4*  
*University of Erlangen-Nürnberg*  
rrkapitz@cs.fau.de

Hans P. Reiser  
*LaSIGE*  
*University of Lisboa*  
hans@di.fc.ul.pt

## Abstract

*Replicating off-the-shelf Java applications is difficult due to the inherent non-determinism of standard Java libraries and multithreading. We propose an OSGi-based architecture that makes applications deterministic at deployment time.*

## 1. Introduction

State machine replication is a fundamental concept to provide fault tolerance. All replicas start in the same state, receive the same input in identical order, have a deterministic behaviour, thus produce identical output and maintain a consistent state. Replication infrastructures typically provide the replication logic in a generic way, separated from the application logic. In practice, however, Java code frequently is non-deterministic. Changing a replica implementation in order that it fulfils the determinism requirements is an intrusive operation, not orthogonal to the application logic. Examples of non-determinism are time access, random numbers, and access to external resources.

It is cumbersome to manually find the respective code sections and replace them with deterministic operations. Previous work handles some cases by intercepting system library calls [1], but this approach is not able to handle non-deterministic behaviour, e.g., of Java class libraries. Even more problematic is that most Java services rely on multithreading. In this case, replica state may depend on the local scheduling of threads, which is not deterministic across machines. Executing a single request at a time is a simple solution that ensures determinism, but it implies reduced performance and may even enforce a complete service redesign if a service implementation based on threads is to be replicated. Alternatively, multithreaded execution can be made deterministic by adequate scheduling support, either by modifying the system scheduler, or by using application-level scheduling [2].

We propose the *DOSGi* approach, which provides instant determinism by automatically intercepting non-deterministic operations during deployment of Java services on the basis of an off-the-shelf OSGi infrastructure [3]. The key contribution of this paper is to show how non-deterministic

code can be dynamically replaced by a deterministic version. The approach is transparent to the application developer (e.g., the application code may use `java.util.Random` to create random numbers), does not require a modified JVM or operating system, and works automatically without user intervention at service deployment time. In a *DOSGi*-enabled distributed infrastructure, a service can be deployed dynamically and instantaneously be made deterministic, allowing the infrastructure provider to offer “replication as a service”.

## 2. Basic architecture of *DOSGi*

The vision of *DOSGi* is to make services deterministic on-the-fly at deployment time. *DOSGi* is embedded in an OSGi system and makes use of a replication framework, which is able to replicate services without further precautions.

Technically, *DOSGi* relies on the OSGi component system for service life-cycle management. OSGi uses components (called *bundles*) that export/import functionality (*Java packages*) to/from other bundles. When a bundle is installed, the OSGi framework *wires* the bundle by resolving these package dependencies. *DOSGi* assumes that an application is composed of multiple bundles and uses OSGi, its code loading facilities, its service abstraction, as well as its support for fine-grained interception, application replacement, and application modification to eliminate non-determinism.

All parts of an OSGi implementation that the replication and determinism mechanisms depend on are deterministic. Non-deterministic functionality such as garbage collection and caching does not influence the mechanisms presented here. In particular, service wiring is deterministic under the condition that the same set of bundles were installed in identical order on all nodes [3].

Figure 1 sketches the architecture of *DOSGi*. It consists of the OSGi run-time system, the replication framework, and our fault-tolerance bundle. The latter has two functionalities. First, it enables the framework to host and instantiate replicas. This is done by the `Factory` bundle that allows services running on other *DOSGi* instances to start new replicas at the current node. It takes care of loading, installing, and starting the required bundles first, and of starting the replica afterwards. The second key entity is the `Rewriter` bundle that comes with the *installation hook*. The installation hook is invoked each time a new bundle is

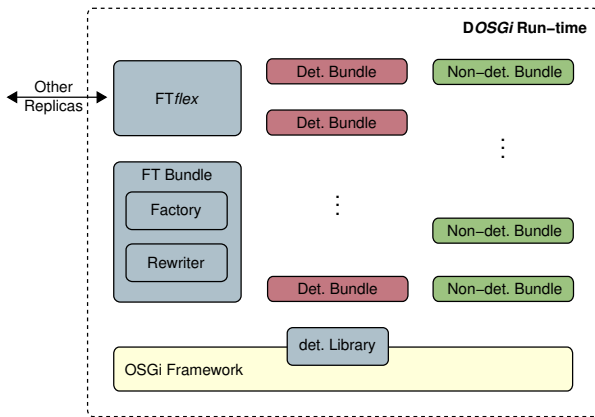


Figure 1. Run-time Set-up of a *DOSGi* Instance

installed in OSGi. It rewrites the code of the bundle so that it makes use of deterministic functions only. Furthermore, it changes the bundle manifest so that the packages that provide the deterministic functions are imported.

### 3. Achieving determinism with *DOSGi*

We use two orthogonal approaches to make existing bundles deterministic: First, loading of custom classes allows substituting non-deterministic code, such as the `java.Math.Random` class and the core `hashCode()` method, with a deterministic replacement. Second, dynamic byte-code rewriting aims at eliminating non-determinism caused by concurrent execution of multiple threads.

#### 3.1. Loading of custom classes

The main source of non-deterministic data is external to the replicated application. The application obtains this data (random numbers, time stamps, etc.) using method calls into the Java library. Yet, it is not possible to just replace the Java libraries using the `bootclasspath` command-line argument, as the OSGi implementation and other non-replicated services running in the same JVM depend on them.

Our two-step approach first constructs a copy of the Java standard library that is 100% deterministic. In this copy, all implementations of non-deterministic methods have been replaced by deterministic versions that also make use of properties of the replication framework. All classes located in one of the `java.*` packages in the original version are placed in `dosgi.java.*` in the copy. In the second step, bundles that will be replicated are changed to make use of the deterministic library instead of the standard library. *DOSGi* achieves this by patching the byte code of the service. The `Rewriter` analyses all classes and methods used by the service and changes entries that

refer to a `java.*` entity to `dosgi.java.*`. In case a bundle imports classes from other bundles that have not yet been made deterministic, a shadow copy of that bundle is created in which all non-determinism has been replaced. It is important to realise that it is not possible to patch the bundle directly, as it might be used by other bundles that do not require determinism and thus are not linked to the replication infrastructure.

During this replacement process, as well as when building the customised library, special care has to be taken for method interfaces that use `Object` as a parameter or return value. This requirement is caused by the fact that arrays may be passed as `Object`. Thus, changing the parameter type makes the methods unusable for arrays. In addition to `Object`, some other classes require special treatment [4]. This is mainly due to the fact that those classes come with special semantics. Our replacement for `Thread` still has to be of type `Thread` in order to use it as a thread. The same holds for exceptions and undeclared exceptions.

#### 3.2. Deterministic concurrent execution

Multithreading is the second source of non-determinism. Concurrent state updates depend on the relative order of synchronisation operations. These operations cannot be intercepted by loading custom classes in case of `synchronize`, `wait()`, and `notify()` statements. Instead, the code patching phase substitutes these statements with calls to our deterministic application-level scheduler [2]. This scheduler uses an algorithm that guarantees that state modifications of concurrent threads remain deterministic.

### 4. Conclusions

The *DOSGi* approach enables transparent on-the-fly determinism of Java applications. This simplifies the replication of such services. The approach applies two novel concepts: transparent class substitution and byte-code patching for substituting synchronisation statements.

### References

- [1] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Enforcing determinism for the consistent replication of multithreaded CORBA applications," in *SRDS*. IEEE, 1999, pp. 263–273.
- [2] J. Domaschka, F. J. Hauck, T. Bestfleisch, H. P. Reiser, and R. Kapitza, "Multithreading strategies for replicated objects," in *Middleware*. Springer, 2008, pp. 104–123.
- [3] OSGi Alliance, "OSGi Service Platform: Core Specification, Release 4, Version 4.1," 2007.
- [4] M. Factor, A. Schuster, and K. Shagin, "Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach," in *OOPSLA*. ACM, 2004, pp. 288–300.