Consistently Applying Updates to Compositions of Distributed OSGi Modules *

Jan S. Rellermeyer Michael Duller Gustavo Alonso

Systems Group, Department of Computer Science,
ETH Zurich, 8092 Zurich, Switzerland
{rellermeyer, michael.duller, alonso}@inf.ethz.ch

Abstract

Updating software at runtime is a challenge that covers various aspects of software design and runtime systems. The OSGi Alliance has proposed and standardized a runtime system for composing Java applications out of modules, the OSGi Framework. The possibility to update modules at runtime and thereby dynamically change the application has been an intrinsic design decision of the framework architecture. With recent approaches to extend the OSGi model from single Java virtual machines to distributed systems, however, updates no longer only affect a single machine in the system. The specifications of OSGi and the upcoming proposals for distributed OSGi services do not answer the question how to consistently apply updates in such environments. In this paper, we explore a solution based on our R-OSGi system. We show how to extend the existing (local) OSGi update mechanism to consistently apply updates to multiple nodes of a distributed OSGi application.

Categories and Subject Descriptors K.6.m [*Management of Computing and Information Systems*]: Miscellaneous; D.3.3 [*Programming Languages*]: Language Constructs and Features; C.2.4 [*Computer-Communication Networks*]: Distributed Systems

General Terms Design, Management

Keywords OSGi, Modules, Updates, R-OSGi

* The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems NCCR-MICS, a center supported by the Swiss National Science Foundation under grant number 5005-67322.

HotSWUp'08, October 20, 2008, Nashville, Tennessee, USA.

1. Introduction

Modular and component systems have gained tremendous momentum in recent times. They finally facilitate the vision of code reusability, which once drove the adoption of object-oriented languages. Furthermore, they appear ideal for building extensible systems that can expose configurable behavior without requiring constant restarts of the entire software. Instead, modules can be dynamically added and other modules can begin to interact with the extensions provided that they are designed to do so. Once uninstalled, the system falls back to the original behavior. These modular architectures are in particular appealing for long-running systems such as application servers. However, they also pose the problem of life cycle management of software modules. One particular challenge is how to apply software updates to the modules. There is a clear demand for updating at runtime and with minimal interruption for systems with high availability requirements. However, applying updates at runtime in a consistent way is particularly intricate due to the fact that modules typically have dependencies between each other. Hence, updating a module can have impact on other parts of the system.

Among the competing frameworks for building applications out of modular components, OSGi [10] is the most advanced in terms of runtime support for software updates. OSGi builds atop the Java virtual machine and adds a layer for running and maintaining software modules. Applications for OSGi have to take into account the dynamism of module compositions, e.g., that single modules can be removed or updated to a newer version at any time, and must react accordingly. The runtime itself facilitates consistent updates by deciding if an update is *safe*—no other modules depend on a piece of code which is updated—or *unsafe*—there are such dependencies that *can* possibly lead to an incompatibility after the update.

Indeed, this behavior is a coarse-grained approach to the update problem. It does not take into consideration the effects of updates on type safety (discussed, e.g., in [8]) or interface compatibility (discussed, e.g., in [3]), which can particularly occur through partial updates. Instead, the model aims at replacing entire modules by new versions and deal-

ing with the resulting effects on other modules which are coupled through dependencies. In this sense, OSGi can be considered to be a minimalistic runtime system for more sophisticated and fine-granular approaches to software updates.

With the currently ongoing efforts to extend the OSGi standard towards distributed systems (OSGi RFC 119) [9], the problem of updates, however, can no longer be solved by updating single nodes in the network individually. Instead, such distributed compositions require consistent updates across different nodes, following the paths of the distributed module dependencies. In this paper, we discuss the update strategies for local OSGi frameworks and present a solution that guarantees the same consistency when applying updates that affect multiple distributed OSGi frameworks.

2. The OSGi Framework

In OSGi, modules (called BUNDLES) are conventional JAR files which have additional meta-information in the manifest. Most importantly, bundles explicitly declare their package imports from other bundles and which of their own packages they are willing to expose to other bundles. The OSGi runtime (called the FRAMEWORK) can thus load each bundle through a separate classloader and delegate the access of shared code imported from other bundles. In contrast to traditional Java applications, which operate on a flat classloader hierarchy, the OSGi framework creates a partly connected graph of dependent classloaders.

Thereby, OSGi is able to give the user of the framework full control over the lifecycle of the single bundles. For instance, the framework facilitates adding new bundles at runtime as well as removing bundles, which restores a previous state of the system by disposing of the corresponding bundle classloader. In order to enable bundles to deal with the dynamism and the runtime changes, OSGi issues events of various types to subscribed listeners.

Besides sharing packages, which causes a tight coupling among bundles, OSGi provides the construct of services, which enables loose coupling. A service is a Java class that implements one or several interfaces. Services are registered in a central service registry and other bundles can use the names of the interfaces for a service lookup.

Great emphasis has been put on providing ways of consistently updating applications running atop an OSGi framework. The framework internally keeps track of which bundles are exporting packages and which bundles are consuming these exported packages through imports. Delegations between exporting and importing classloaders are established when resolving the bundles and tracked as *wires*. Once wired, a delegation remains immutable[1]. The rationale behind the OSGi update strategy is to keep bundles in a functional and consistent state. Therefore, actions which indirectly affect running bundles by changing shared code are deferred until a framework user explicitly triggers a package refresh[2]. The two actions which potentially lead to such changes are the uninstallation of bundles and bundle updates, as they both remove (old) packages.

2.1 Uninstalling Bundles

The uninstallation of a bundle is only considered safe if no exported package is in use by other bundles at the point in time when the action occurs. This is trivially true for bundles not exporting any package. If the action is not safe the classloader has to be marked for removal but the packages will remain accessible to existing consumers until a package refresh for all or only the affected packages is triggered. All packages not in use are removed from the system by removing the corresponding package information from the internal package database of the framework. Subsequently, bundles arriving after the uninstallation of a bundle no longer be wired to an uninstalled bundle any more.

2.2 Updating Bundles

Bundle updates follow the same pattern. Packages which are not exported or are not used by downstream consumers are immediately updated. For packages in use, the classloader hosting the original version of the bundle has to remain accessible for exactly these packages. Newly appearing consumers of packages which were previously not in use will be wired to the classloader serving the new code and hence they will *see* the new version. Figure 1 shows such an update where packages 1 and 4 are in use by other bundles and therefore preserved after the update.

Regardless of the safeness of the update, an UPDATED event is signaled to those bundles which have opted in for being informed about state changes by registering a BUNDLELISTENER with the framework.

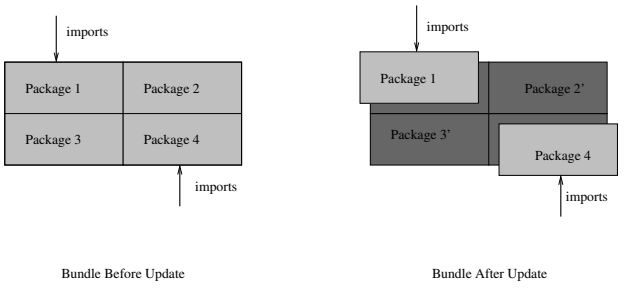

Figure 1. Update of a Bundle with Exported Packages in Use

2.3 Refreshing Packages

Packages can be explicitly refreshed through the PACKAGEADMIN service, which is a service described in the core specifications [10]. Even though this service is optional,

[1] Except when rewired through a package refresh, which, as discussed later, requires explicit intervention

[2] In Release 3 of the specifications. Release 4 allows multiple versions of packages to coexist in disjoint classloader spaces

most existing OSGi framework implementations provide it by default. A refresh can be triggered by any bundle[3] installed on the framework. Examples of such a bundle are a console or a GUI which facilitate user intervention.

With each refresh, an initial set of bundles that need to be refreshed is passed as an argument to the package admin. This can be either a fixed array of bundle objects, or `null`, which is defined to be all bundles in the system. From this initial set of packages, the package admin service then determines the exported packages which have been marked for removal or which have a deferred version update. For this (smaller or equal) set of packages for which a refresh actually has an effect, the package admin recursively builds the transitive closure of all bundles that are dependent due to an incoming wire; i.e., if a bundle imports a package from a bundle that is subject to a refresh, it gets refreshed as well and thus all its exports are also marked to be refreshed. The result is a set of packages (and thereby a set of exporting bundles) thats are affected by the refresh action.

To effectively release all references to old packages, the refresh process itself first stops all affected bundles in reverse start level and installation order. Exceptions occurring during this process of stopping the bundles are signaled through framework events to subscribed listeners but otherwise ignored. After the last bundle has left the active state, stale packages (as results of uninstallations) are removed by disposing of the classloader, and deferred package updates are applied by swapping the classloader reference to the classloader providing the updated version of a bundle. Subsequently, all bundles which have not been uninstalled are restarted corresponding to their startlevel and original installation order. The result is a system which now entirely operates on the refreshed packages. Any reference to uninstalled packages or previous versions are dropped. However, bundles that previously were active may no longer be resolved because required package dependencies could have been uninstalled.

3. Distributed OSGi

Recently, there have been efforts to extend the OSGi model to interconnect services running on distributed OSGi frameworks. Systems like NEWTON [11] or the OSGi-internal RFC 119 describe solutions which operate on the service layer and provide the remote access to services over the network. These approaches are limited because they solely operate on the level of services and ignore the bundle layer. They require all clients of remote services to have the corresponding service interfaces and custom types referenced by the interfaces to be available. In terms of code updates, this causes a significant problem. Updates on the remote service bundle remain undetected by clients. However, they can lead to inconsistencies within the distributed service composition when the interface or types referenced by the interfaces undergo a change.

In contrast, R-OSGI [14] explicitly deals with the module layer. R-OSGi embeds the service proxy into self-contained proxy bundles. In addition to shipping the service interface to the client, R-OSGi uses static dependency-analysis on the service to determine the minimal set of classes of a service bundle that would make the interface resolvable on an arbitrary client device. These classes are additionally shipped to the client, materialized in the proxy bundle, and thus provided to the client runtime (*type injection*). Thereby, a client of a service that is accessing a service proxy *sees* exactly the part of the remote bundle which is exposed by the remote service (Figure 2).

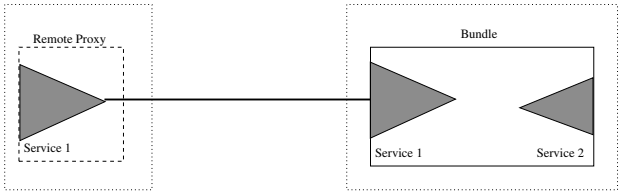

Figure 2. Remote Proxies for OSGi Services with R-OSGi

Furthermore, the proxy bundle partly inherits the package import behavior. Out of the original package imports, the proxy must import exactly those packages that are required to resolve the classes that are injected into the proxy bundle. Therefore, the package import set of the proxy is a subset of the import set of the original bundle. The proxy exports exactly those packages that were exported by the original bundle and that have at least one member in the form of a class injected into the proxy bundle.

As a consequence, the proxy bundle generated on the client side can have dependencies that have to be resolved in order to have a working service. In the recent development version, R-OSGi does not assume that all bundle dependencies of a service are present on each client node. If a package import is not resolvable on a client, R-OSGi creates a clone of the bundle that resolved the package import of the original bundle to mimic the behavior of a single local framework (Figure 3).

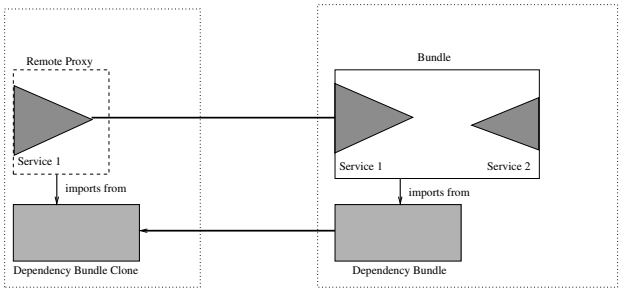

Figure 3. Update Problem for Managed Proxy Dependencies

[3] When running with permissions enabled, only bundles with appropriate permissions can trigger a refresh.

However, this approach poses the challenge of consistently handling updates occurring on the original service bundle. The OSGi framework itself is not capable of dealing with distribution, hence, it does not provide any support for distributed bundle updates.

4. Distributed Updates

Distributed updates are more complex than local updates. First, local updates are handled by the framework which has full knowledge about package versions, wires, etc. The distributed update is handled through the distribution software which itself is just a bundle and has no direct access to the internal state of the framework. Second, local updates happen on the granularity of entire bundles. The distribution software, however, primarily operates on the granularity of services. Therefore, it generates stripped down proxy bundles that only provide the service interface and necessary type injections, as discussed in the previous section. As a result, updates to the service bundle cannot be directly applied to the proxy bundles. The distribution software has to handle this case specifically. Furthermore, as discussed in section 2.2, the update can potentially be applied by the local framework in two steps, the immediate and the deferred part. The middleware has to mirror both parts at the right time to the client nodes. Third, consistency is harder to achieve when multiple peers are involved. Atomicity is, as we will show, not possible without additional knowledge about the update.

4.1 Detecting the Update

Since the distribution middleware sits on top of the framework as a regular application, it can only detect bundle updates through the events they generate. Unfortunately, the events signaled by the framework do not give sufficient information about the exact update procedure. After a successful update of a bundle, a BUNDLEEVENT of type UPDATED is issued. As a payload, it contains the information on which bundle the update happened but not what the precise effect of the update is. The whole picture has to be constructed by correlating such events with feedback information gathered through interaction with the package admin. The package admin provides an interface for retrieving information about exported packages of a bundle and their versions. The prerequisite for detecting an update, however, is that the distribution middleware has sufficient information about the previous state of the bundle to determine the set of packages that have been changed. However, this assumes that updates follow a strict versioning discipline, i.e., the package version is actually incremented in the metadata whenever a class of the package has changed. The other possibility would be to track the state of individual classes by calculating hashes over the entire bytecode or certain parts of the class as done for serial version UIDs in the Java Serialization Protocol [17]. Since updates which do not change the package versions can have side effects (e.g., breaking dependent bundles) even in the case of a local OSGi framework, the R-OSGi middleware was not implemented to support per-class tracking of updates.

4.2 Preparing the Update

Unfortunately, the middleware has no access to the bundle that caused a local update. This version might or might not be persisted by the framework in its private storage but there is no reliable and framework implementation-independent way of accessing this data. With the information about changed packages, however, the middleware can reconstruct an update bundle. The corresponding bytecode can be retrieved from the classloader. For the state of the bundle after the package refresh, this is easier because all packages and classes are then provided by the same classloader. For reconstructing the state right after the update event, there are potentially two classloaders that refer to the bytecode of the classes. One is the old classloader which serves the deferred packages and one is the new classloader which serves the updates version.

If the update affects dependency bundles, this generated update bundle can be directly applied. For the service bundle, the generated update bundle contains more classes than the proxy bundle. However, the type injection property is an invariant that is expected to hold after the update. Hence, the same code analysis algorithm can be used that is already part of R-OSGi for determining the type injection set. Applied to the new state of the bundle after update, the code analysis generates the corresponding update bundle.

4.3 Applying the Update Consistently

In order to apply the updates atomically, the distribution software has to take on the role of the package admin on the corresponding client peers. It has to execute the algorithm described in 2.3 but introduce a distributed consensus between the stop and start phases that involves a possible rollback. Thereby, the semantics of a local OSGi update is consistently mirrored to a set of client peers.

However, even with this effort, applying the update atomically in a global sense is not possible. The distribution software can only react to events and the update event is generated after the update has already been successfully applied locally. It therefore can only observe an update but not influence (e.g., abort or delay) it any more. There are two ways of fixing this behavior and turning the entire distributed update into a single atomic transaction. One is to alter and intercept the operation of the framework. If the distribution software can influence the outcome of the initial update, it can achieve a global atomic update semantic. However, non-invasiveness against the standardized framework implementations was a clear design decision for the R-OSGi middleware. The other approach is to apply the updates through an external distribution-aware interface and not directly through the local framework. In past work, we developed a deployment tool [13] for R-OSGi which facilitates the decomposition of

an application within the Eclipse [4] IDE and the deployment to a set of machines through a graphical user interface. The same infrastructure can be used to apply updates to bundles to all affected peers atomically because it enables the distribution software to act before the update is applied on the local framework.

5. Related Work

Several authors have discussed the challenges of runtime updates in different domains in a number of research projects, e.g., [7], [5], [6], [16], [2].

The Ginseng system [8] allows to update software without stopping and restarting it. It generates dynamic patches that can be applied to a running application with the help of the runtime system. We do not require updates to fully running applications but instead leverage the standardized OSGi model to only stop and restart those parts of the application consistently that are actually affected by the update.

The author of [12] identifies software evolution and thus updates to software at runtime as an important property of distributed applications in particular. She proposes to consider updates as cross-cutting concerns and therefore express them as aspects which are woven into the application at runtime. We also see handling of updates as a concern of the middleware layer and we agree that proper means and support for software evolution are a necessity for distributed systems.

In contrast to the approach presented in this paper, the authors of [1] research systems for which it cannot be assumed that updates can happen for all nodes at once. Instead, the authors discuss a system that gradually updates nodes.

Research has also been conducted on using different versions of code at the same time and letting old and new code interact [15]. Coexisting versions of code can also arise in our approach when old packages are still in use by other bundles. However, the typical focus of applications written in OSGi is to eventually remove all old uses of old versions of packages and only execute one version of code throughout the system.

6. Conclusion

We introduced OSGi's module layer and its code dependency management. Then we presented R-OSGi, which provides interaction with remote OSGi services not only on the service layer but also incorporates the module layer by handling dependencies through type injections and dependency bundle clones. As a result, R-OSGi can track updates to modules and update injected dependencies or dependency bundle clones on remote nodes respectively resulting in consistent updates to compositions of distributed OSGi modules. If perfect atomicity for both remote service clients and local service clients is a requirement, an external update coordinator like the R-OSGi Deployment Tool can be used.

References

[1] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *ECOOP '06: European Conference on Object-Oriented Programming*, July 2006.

[2] A. Baumann, G. Heiser, J. Appavoo, D. D. Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference*, pages 32–32, 2005.

[3] A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdzinski, and F. Mang. Interface compatibility checking for software modules. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, 2002.

[4] Eclipse Foundation. The Eclipse Project. `http://www.eclipse.org`.

[5] 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.

[6] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Programming Language Systems*, 27(6):1049–1096, 2005.

[7] I. Lee. *Dymos: a dynamic modification system*. PhD thesis, The University of Wisconsin - Madison, 1983.

[8] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. *SIGPLAN Not.*, 41(6):72–83, 2006.

[9] E. Newcomer and T. Dieckmann. *Distributed OSGi - External Services and Service Discovery*, 2008.

[10] OSGi Alliance. *OSGi Service Platform, Core Specification, Release 4, Version 4.1*, 2007.

[11] Paremus Ldt. Newton. `http://newton.codecauldron.org`, 2006.

[12] S. C. Previtali. Dynamic updates: another middleware service? In *MAI '07: Proceedings of the 1st workshop on Middleware-application interaction*, pages 49–54, 2007.

[13] J. S. Rellermeyer, G. Alonso, and T. Roscoe. Building, Deploying, and Monitoring Distributed Applications with Eclipse and R-OSGi. In *ETX '07: Fifth Eclipse Technology Exchange Workshop*, 2007.

[14] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed Applications Through Software Modularization. In *Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference*, 2007.

[15] P. Sewell. Modules, abstract types, and distributed versioning. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 236–247, 2001.

[16] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Programming Language Systems*, 29(4):22, 2007.

[17] Sun Microsystems. Java Object Serialization Specification, 1997.