Developing an OSGi-like Service Platform for .NET

Clément Escoffier, Didier Donsez, and Richard S. Hall
Laboratoire LSR-IMAG, 220 rue de la Chimie
Domaine Universitaire, BP 53, 38041 Grenoble, Cedex 9 FRANCE
{clement.escoffier, didier.donsez, richard.hall}@imag.fr

Abstract

The OSGi specification defines a Java-based service platform for dynamically deploying services into networked environments. OSGi technology originally targeted home services gateways, but is now used as a general Java application extensibility mechanism. The main abilities contributing to its growing influence are its support of a dynamic service deployment life cycle and its amenability to remote management. Microsoft's .NET platform, in some ways, improves upon the Java platform, but it still lacks explicit support for building dynamically extensible systems like those made possible by the OSGi framework for Java. This paper presents the results of an effort to create an OSGi-like service platform for the .NET platform.

1. Introduction

The OSGi specification [8], defined by the OSGi Alliance [9], defines a service platform for dynamically deploying services into networked environments. The initial target domain for the OSGi framework was home services gateways, but its target domain has expanded to automotive and mobile telecommunications industries and to Java development in general. The main abilities contributing to its use in these new areas are its support of a dynamic service deployment life cycle and its amenability to remote management. These strengths result from the basic assumptions that service availability is dynamic in nature and that service providers require remote access for management.

The OSGi framework's focus on dynamics is indicative of many current computing trends, such as autonomic [5], auto-adaptive [1], proactive [17], and context-aware [2] computing, that are all trying to manage the complexity of software system execution in dynamically changing environments. Such systems must be flexible enough to cope with situations where, at any moment, required pieces of functionality become unavailable or new pieces of useful functionality are introduced. Distributed systems have long had to deal with similar dynamic availability issues, but this situation is now commonplace within a single computer and even a single process, which is mirrored in the OSGi framework's non-distributed approach.

The main reasons that dynamic availability is now an issue across the board is related to the arrival of platforms, such as Java, that have greatly simplified the process of dynamically loading and executing code and to the success of component- and service-oriented development approaches [14][10]. Since dynamic code loading is now so simple, it is very common for systems to be designed as *dynamically extensible systems* [13], "which can cope with late addition [and removal] of extension without requiring a global integrity check." This meshes nicely with component- and service-oriented development, which both define development building blocks that form perfect units for deployment and discovery in extensible systems.

The OSGi framework extends the standard Java platform to provide better support for creating dynamically extensible software systems. Consequently, the OSGi framework has started to score recent successes, such as being adopted by the Eclipse platform [3] as a dynamic plugin engine. While the OSGi framework is largely a Java phenomenon, the advantages that it provides are not completely lost on other communities, such as those interested in Microsoft's .NET platform [7]. While the .NET platform offers some advantages over the Java platform, it still lacks explicit support for building dynamically extensible systems.

This paper presents the results of several attempts to implement a non-distributed service platform like OSGi for the .NET platform. Section 2 discusses the dynamic code loading mechanisms in the Java and .NET platforms. Section 3 describes and critiques alternative realizations of an OSGi-like layer for the .NET platform, followed by the conclusion.

2. Dynamic Code Loading

This section first discusses code loading in Java as well as how OSGi extends the standard mechanisms, then standard code loading in .NET is discussed.

2.1 Code Loading in Java

The Java platform is based on a virtual machine that abstracts the underlying operating system and makes it possible to safely run programs across heterogeneous computing platforms. The Java virtual machine [16] interprets portable byte code delivered in the form of class files. Class files are dynamically loaded into the virtual machine following an explicit search order.

The actual run-time mechanism used to load classes in the virtual machine is an instance of the `ClassLoader` class, which is a special class used by the Java platform to load other classes. [6] All class loaders have a parent class loader, except for the bootstrap class loader. By default, child class loaders delegate class load requests to their parent class loader and only if the parent does not find the class will the child search for the class itself. When a Java application is started, the application's main class is loaded with a default application class loader instance, which ultimately delegates to the boot class loader. It is possible for applications to provide custom subclasses of `ClassLoader` to perform specialized searches for classes, such as searching remote repositories. It is through class loaders and class loader customization that Java popularized or, at the very least, brought dynamic code loading to the masses.

2.2 Code Loading in Java with OSGi

The OSGi specification improves upon the basic features of the Java platform by defining a lightweight component abstraction, a standard packaging format, a dynamic deployment life cycle, and a service registry for component interaction. The OSGi packaging format, called a *bundle*, is a standard Java archive (JAR) file, with a manifest file that contains metadata concerning Java class packages required by the bundle (*imports*) and Java class packages provided by the bundle (*exports*). Besides the manifest file, a bundle JAR file may contain class and resource files, other embedded JAR files, and native code. Import and export information is expressed in terms of Java class package names and versions.

The OSGi framework ensures that the only classes visible to a given bundle are those contained in its own JAR file and those which it imports. Unlike the standard Java class loading search order, which is hierarchical, the OSGi search order is a graph, which allows it to support a dynamic deployment life cycle for bundles (e.g., install, activate, update, and uninstall) [4].

The standard Java platform also provides a mechanism, called Optional Packages [12], that enables describing dependencies among JAR files. Using this mechanism, it is even possible to download and update JAR files automatically. The approach is less sophisticated and not as flexible as the OSGi approach and, more importantly, is largely static.

In addition to dynamic deployment life cycle management, the OSGi framework also provides a service registry where bundles can publish independent implementations of Java interfaces, called services, and other bundles can search for available service implementations. Services are the only form of direct component interaction supported by the OSGi framework. The use of service interfaces insulates bundles from service implementation details. A reference to a service is a direct reference, using normal method invocation.

2.3 Code Loading in .NET

Microsoft's .NET platform is similar to the Java platform in many respects, but some significant differences do exist. First, the .NET virtual machine [15] was designed to supports multiple programming languages (e.g., C#, J#, VB.NET, and Visual C++ .NET), whereas the Java virtual machine was only designed to support Java, although other languages for it do exist (e.g., SmalltalkJVM, Groovy, Jython, JRuby, and Nice). In .NET, all languages run on the Common Language Runtime (CLR). The CLR uses a portable language format, called the Microsoft Intermediate Language (MSIL), which is analogous to Java's byte code. High-level languages are compiled into MSIL, which is then compiled into the native code of the underlying computing platform at load time, similar to how Just-In-Time (JIT) compiling works in Java. Unlike anything in Java, the CLR is able to retain assemblies in a Global Assembly Cache (GAC) for later reuse and version management. The official .NET platform is from Microsoft. Microsoft also provides a “shared source” implementation, called Rotor [11].

In .NET, applications and their components are packaged into *assemblies*. An assembly is .NET's unit of reuse, versioning, security, and deployment. An assembly is one or more files representing types and resources and is described by a manifest. An assembly manifest is represented in XML and contains information such as version number, natural language, and content hashes. Hashes come into play when calculating an assembly's signature using public key cryptography, which is also referred to as the *strong name* of the assembly and it is used for identification and security purposes. Assemblies can be dynamically loaded, but, unlike Java class packages contained in JAR files, it is not possible to load individual types or classes from an assembly.

A .NET application runs in an isolated execution environment, called an application domain. Application domains are analogous to virtual processes. Several application domains can share the same virtual machine (and thus the same physical process), but they are treated as completely separate. Communication between two application domains must use interprocess communication (IPC), such as .NET Remoting.

Application domains manage the loading of assemblies. By default, an application's constituent assemblies are loaded into a single application domain. Assemblies loaded into different application domain are not directly accessible to each other. An application can create several domains for loading assemblies.

When an application references a type, if the type has already been loaded into the domain, then it is reused. If not, then the runtime searches for the assembly containing the type and loads it. Application domains can also be unloaded, which releases any assemblies loaded into the domain. This is the only way to unload code in the .NET platform, since types in .NET are not regarded as "normal" objects and are not garbage collected like in Java.

The .NET runtime searches for assemblies by first probing the GAC and then the application's directory. Assemblies in the GAC are accessible to all applications, whereas assemblies in an application directory are private to that application. Through special configuration files, an application can modify the assembly search process by defining version redirection rules, additional directories to search, and code bases from where an assembly can be downloaded.

3. An OSGi Service Platform for .NET

The objective of the work described in this paper was to study the possibility of creating a single-process dynamic extensibility framework for .NET, similar to what the OSGi framework provides for Java. In doing so, several approaches were implemented. Each approach was evaluated with respect to the following criteria:

- Ability to dynamically load/unload services and their supporting code and resources.
- Performance of service invocation.
- Controlled access to dynamically loaded code.

It is important to point out that it is unlikely that the OSGi framework's capabilities could ever be ported to the .NET platform in a completely isomorphic way. The reason for this is that there are certain impedance mismatches between the two platforms. For example:

- In Java, the unit of code loading is a class; in .NET, the assembly is the unit of code loading, which may contain many types.
- In Java, the compiler does not record explicit dependencies among classes, which can be arbitrarily resolved at run time; in .NET, assembly dependencies are explicitly recorded at compile time, making them difficult to resolve differently at run time.
- In Java, the class search order is nearly completely customizable via class loaders; in .NET, class/assembly searched order controlled by more sophisticated policies that complicate implementing custom search orders.
- In Java, the deployment unit does not form part of the class name; in .NET, the assembly name forms part of the contained type names, ultimately reducing provider substitutability.

With these criteria and considerations in mind, the following subsections describe and critique various approaches for implementing an OSGi-like dynamic service platform for the .NET platform. Besides the alternative approaches described in the following subsections, some effort was also put into trying to modify the underlying open source CLR implementation to achieve better service platform characteristics. However, this effort was not successful and it appears any effort to do so would be significant. Further, the value of a non-standard CLR is questionable.

Of the four remaining approaches discussed in the next subsections, all use single-file assemblies for simplicity. In each approach, an assembly contains zero or more service implementations. The general packaging approach is to place service interfaces in their own assemblies; thus, assemblies containing service clients and providers depend on service interface assemblies. Also for simplicity purposes, the service registry is simply a list of services where each service entry is a name, an interface type, and an actual service implementation instance.

3.1 Single Application Domain Approach

The first alternative used a single application domain. The single domain contains a special loader assembly that manages service assembly loading for the domain; figure 1 illustrates this architecture.

The main benefit of this approach is that service method invocation is fast, since it is just a local method call. While it is possible to dynamically load service assemblies, it is not possible to dynamically unload them. The .NET platform does not support unloading an individual assembly from an application domain. To unload an assembly, it is necessary to unload the entire application domain, which in this approach is equivalent to restarting the entire system. This issue consid-

erably limits this approach as a way to implement an extensibility framework.

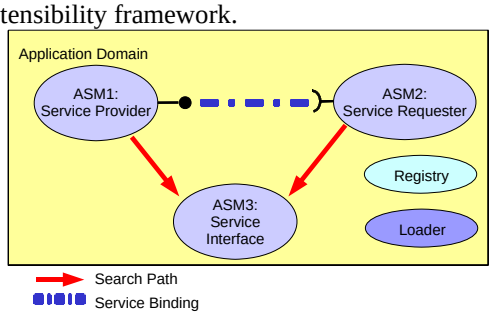

Figure 1. Single application domain approach.

3.2 Multiple Application Domain Approach

The second alternative used multiple application domains. Each service assembly is loaded into its own application domain. The loader assembly also runs in its own domain. Figures 2 illustrate this architecture.

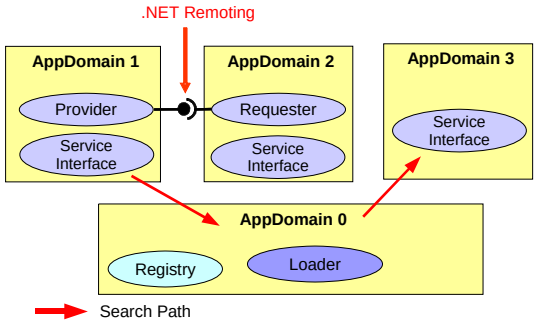

Figure 2. Search path for the multi-domain alternative.

The loader also manages the set of available resources in each domain. When an assembly tries to load a resource it delegates this request to the loader. The loader then searches other application domains. If the resource is available in another domain, the loader transfers the resource to the requesting domain; in the case of a service interface, the loader creates a proxy and copies it into the requesting domain. With this approach it is possible to provide better insulation among assemblies by following a pattern of separating a service into two assemblies: the service interface and the service implementation. By doing so, client assemblies will only load the service interface assembly into their application domain when accessing the service and will not have access to other public classes in the service implementation assembly.

Despite the advantages of this approach, it has an undeniable disadvantage. Since an application domain is an isolated execution environment, communication between application domains requires the use of an IPC mechanism. Inter-application domain communication is handled by .NET Remoting, which is quite expensive in terms of performance. The main overhead penalty is incurred due to serialization/de-serialization of exchanged objects during method invocation. The difference in performance is dramatic when compared to local method invocations. It is possible that future IPC mechanism could mitigate this overhead, which might render this approach usable.

The loader manages loading/unloading service assemblies and creating/unloading application domains for each service implementation.

A final issue in this approach is that application code must be aware of the modified resource delegation search process and be coded explicitly for it.

3.3 Shared Application Domain Approach

The third alternative used a somewhat hidden feature of the .NET platform, called the *shared domain*. The shared domain is not really an application domain, since it is not an execution environment; however, it is possible to load assemblies into the shared domain. Assemblies loaded into the shared domain are called *neutral domain assemblies* and their JIT compiled code is shared among all application domains within the physical process. Neutral domain assemblies require an actual application domain to execute. Figure 3 shows how the service interface is copied from the shared domain into other domains; the searching and copying is performed automatically by the CLR. By default, the core .NET assembly, containing basic types, is loaded into the shared domain because it is used by all .NET applications. The shared domain improves performance and resource consumption.

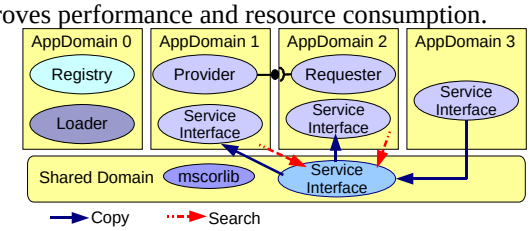

Figure 3. Search path in the Shared Domain alternative.

This third alternative forces the loading of some assemblies into the shared domain, but ultimately this approach is similar to and potentially worse than the second approach. It still suffers from invocation performance issues, since domain neutral assemblies are still conceptually copied into each referencing application domain, which again results in the use of IPC for service method invocation. To make matters worse, domain neutral assemblies are accessible to all other assemblies and can never be unloaded.

3.4 Hybrid Approach

Another possible approach, not implemented, is to combine the first two alternatives. In such a hybrid approach, highly coupled services could be placed into one domain to allow local service invocation. Service calls would then be invoked in one of two ways: by direct method invocation if the service object is in the same application domain as the caller or via .NET Remoting if the service object is in a different application domain than the caller.

In such an approach, each application domain has a local service registry, where all local services can be found. The local service registry must also coordinate with a global service registry for inter-domain service discovery. This architecture also opens up two new possibilities:

- Not all services must be globally shared among application domains and
- Multiple versions of the same service may exist in the service registry at the same time.

These two new possibilities add subtle complexities to the overall model, since errors will occur if a client encounters an unexpected version of a type. This gives the extensibility framework the responsibility of ensuring type-space consistency for clients; this form of support for multiple service versions and type-space consistency is also under consideration for the next version of the OSGi specification.

Updating or uninstalling services in this approach would still require that the containing application domain be unloaded. The impact of such operations on the overall system is a trade off between communication performance and dynamic resiliency, which results from deciding whether service assemblies should be loaded into separate or the same domain. As a result, the value of this proposed approach is at the mercy of the algorithm used to place assemblies.

4. Conclusion

The conceptual dynamic service platform defined by the OSGi specification is gaining widespread acceptance. The ability to dynamically deploy and remotely administer services is feeding this acceptance. The OSGi framework has outgrown its original target domain of home services gateways and is now targeting applications in automotive, mobile telecommunications, and Java development in general. Despite the fact that the OSGi framework is largely a Java phenomenon, the concepts it embodies are also of interest to developers on Microsoft's .NET platform.

This article presented and critiqued several approaches to implementing an OSGi-like framework for the .NET platform. The results indicate that .NET lacks certain capabilities to create a similarly flexible and lightweight service platform as the OSGi framework. The inability of .NET to unload assemblies from within an application domain and the poor performance of inter-application domain communication, result in an inadequate dynamic service platform.

10. References

[1] L. Andrade and J.L. Fiadeiro. "*An Architectural Approach to Auto-Adaptive Systems*," 22nd Int'l Conference on Distributed Computing Systems Workshops, July 2002.

[2] A.K. Dey and G.D. Abowd. "*Towards a Better Understanding of Context and Context-Awareness*," Workshop on the What, Who, Where, When and How of Context Awareness at CHI, April 2000.

[3] O. Gruber, B.J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. "*The Eclipse 3.0 Platform: Adopting OSGi Technology*," IBM Systems Journal, Volume 44, Number 2, 2005.

[4] R.S. Hall. "*A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks*," Proceedings of the 2nd International Working Conference on Component Deployment (CD 2004), May 2004.

[5] J.O. Kephart and D.M. Chess. "*The Vision of Autonomic Computing*," IEEE Computer, January 2003.

[6] S. Liang and G. Bracha. "*Dynamic Class Loading in the Java Virtual Machine*," Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'98), October 1998.

[7] J. Ntuba. "*Design and Implementation of an OSGi Service Architecture for the .NET Platform*," Masters Thesis, Free University Berlin, July 2004.

[8] Open Services Gateway Initiative. "*OSGi Service Platform Version 3*," http://www.osgi.org, March 2003.

[9] OSGi Alliance. "*Web site*," http://www.osgi.org, 2004.

[10] Z. Stojanovic and A. Dahanayake. "*Service-Oriented Software System Engineering: Challenges and Practices*," Idea Group Publishing, 2005.

[11] D. Stutz, T. Neward, and G. Shilling. "*Shared Source CLI Essentials*," O'Reilly Media, Inc., March 2003.

[12] Sun Microsystems. "*The Java Extension Mechanism (for Support of Optional Packages)*," Sun Microsystems Java Documentation, 2002.

[13] C. Szyperski. "*Independently Extensible Systems - Software Engineering Potential and Challenges*," Proceedings of the 19th Australian Computer Science Conference, 1996.

[14] C. Szyperski. "*Component Software: Beyond Object-Oriented Programming*," ACM Press/Addison-Wesley Publishing Co., 1998.

[15] T.L. Thai and H. Lam. "*.NET Framework Essentials*," O'Reilly Media, Inc., August 2003.

[16] B. Venners. "*Inside the Java Virtual Machine*," McGraw-Hill Companies, 1997.

[17] R. Want, T. Pering, and D. Tennenhouse. "*Comparing Autonomic and Proactive Computing*," IBM Systems Journal, Volume 42, Number 1, 2003.