PAAL, Stefan
KAMMÜLLER, Reiner
FREISLEBEN, Bernd

Separating the Concerns of Distributed Deployment and Dynamic Composition in Internet Application Systems

Publiziert auf netzspannung.org:
http://netzspannung.org/about/technology/
24.06.2004

Erstveröffentlichung**:** Distributed Objects and Applications (DOA 2003).
LNCS 2888. Catania, Italy: Springer, 2003, S. 1292-1311.

Separating the Concerns of Distributed Deployment and Dynamic Composition in Internet Application Systems

Stefan Paal [1], Reiner Kammüller [2], Bernd Freisleben [3]

[1] Fraunhofer Institute for Media Communication
Schloss Birlinghoven, D-53754 Sankt Augustin, Germany
stefan.paal@imk.fraunhofer.de

[2] Department of Electrical Engineering and Computer Science, University of Siegen
Hölderlinstr. 3, D-57068 Siegen, Germany
kammueller@pd.et-inf.uni-siegen.de

[3] Department of Mathematics and Computer Science, University of Marburg
Hans-Meerwein-Strasse, D-35032 Marburg, Germany
freisleb@informatik.uni-marburg.de

Abstract. The Internet is currently evolving from a *global information network* into a *distributed application system.* For example, some Internet applications are based on executing remote services which have been previously installed on possibly multiple Internet nodes, whereas parts of other Internet applications are dynamically moved from several remote nodes to be executed on a single node. In this paper, we focus on the related problem of how the parts of an Internet application that have been independently deployed on multiple Internet nodes can be transparently located, seamlessly retrieved and dynamically composed on a particular node by request. We propose a novel deployment and composition approach using so called *modules* and *module federations* and show how to separate the logical application composition from the physical module deployment. The realization of our proposal in Java and C++ is presented and the use of the approach in ongoing research projects is demonstrated.

1 Introduction

Originally, the Internet was supposed to interconnect spatially distributed computing nodes and serve as a *communication* medium to exchange data among them. However, during its evolution, the Internet slowly turned into an *interconnection* medium that was used by some Internet applications to link Internet nodes on the application level. Each Internet node hosting a web server became part of a certain kind of *service federation*, namely the WWW, and users got the illusion of a *global information network* in which all participating nodes are seamlessly linked together and the web browser is a universal interface to this network [1]. With the advent of *web services* [2] in recent years, Internet nodes offer customized services

which can be remotely accessed similar to web servers [3]. The invention of web services also promoted new kinds of Internet application systems, so called *application servers*, which are able to host custom Internet applications instead of fixed applications. Moreover, today various Internet applications are distributed and interact across multiple nodes, as in the case of the *grid computing paradigm*. From this point of view, the Internet finally turned into a *cross-platform application environment* and the Internet nodes have become part of *distributed Internet application systems* [4, 5]. In the depicted contexts, Internet applications are typically statically deployed and appropriately configured on each node by the site administrator. However, it is neither always possible nor desired to deploy and setup an application in advance on the node where it is to be executed. Thus, an application is not longer installed on each node but rather deployed into an application repository. From there, it is automatically downloaded onto the target node and appropriately configured before the application is started, as e.g. in *Sun Java Web Start* [6] or *Netx* [7]. In other scenarios, applications are not longer deployed in terms of a single executable but are dynamically composed of smaller parts like libraries or components, as e.g. in the case of *Sun Enterprise Java Beans (EJB)* [8]. Furthermore, similar to orchestrating web services at runtime (which, however, typically remain on the remote node) [9], an Internet application may also be flexibly composed by downloading services from different providers dynamically to perform additional tasks on a single node as in the case of *Jtrix* [10]. Finally, already running applications may want to travel across various nodes (such as software *agents)* or they have to migrate from one host to another. While migration itself already raises many questions regarding saving and restoring object states and application contexts, a major problem is *code mobility* in terms of deploying, configuring and composing an application [11].

To summarize, there are scenarios where Internet applications have to be arbitrarily deployed and executed across various Internet nodes. This leads to a particular need for dynamically locating, retrieving and installing an application from and on certain nodes by request. Moreover, an application itself may be composed of smaller parts, which also touches a major problem in *component-based software engineering (CBSE)*, namely the negotiation and gluing of unknown and originally incompatible components [12]. However, while this indeed is an important issue in an open world scenario, we think that Internet applications are not typically composed of unknown and inherently insecure code fragments but rather of familiar and trustable elements. Thus, in the following we explicitly assume that the composition of related Internet applications is characterized by *selecting* an appropriate component out of a group of compatible components and not by *gluing* possible incompatible components. Therefore, the focus of the approach presented in this paper is the clear separation of component deployment, platform configuration and application composition. We propose a novel deployment and composition approach using so called *modules* and show how they can be used to separate the deployment configuration of the current Internet node from the composition configuration of the Internet application. Along with that we introduce remote module repositories and organize them in so called *module federations* which enable to transparently deploy and query modules in a distributed application system and to dynamically retrieve them from various remote nodes by request.

The paper is organized as follows. In section 2, we discuss the features and requirements of distributed Internet application systems regarding deployment and composition issues and consider existing solutions. Section 3 presents our approach to separating the deployment and composition configuration and illustrates its realization in Java and C++. The application of the approach is demonstrated in section 4. Finally, section 5 concludes the paper and outlines areas for further research.

2 Distributed Internet Application Systems

In this section, we address two of the fundamental problems concerning code mobility, namely deployment and composition of Internet applications.

2.1 Deployment

A basic task of software development is the deployment of an application. In a single managed runtime environment an application can be deployed in a simple manner, but in a distributed Internet application system with multiple involved and independently managed nodes the deployment task is more complex. In this context, we define a *module* as a deployable unit which may contain different code resources and can be separately distributed. The following issues have to be addressed with respect to deployment in distributed Internet application systems.

Managing Different Variants. A module typically exists in different variants, e.g. debug or release version, single-thread or multi-thread capability and so on. We assume that an appropriate module variant is selected matching given attributes. Furthermore, it is likely in a distributed Internet application system that every variant is retrieved concurrently by possibly different nodes. Consequently, a basic feature is the deployment of the same module in different variants and its simultaneous use and provision.

Distributed Deployment. An application on a standalone platform can only retrieve locally deployed modules which typically do not vanish. In contrast, an Internet application has to retrieve modules from remote module repositories which may disappear by chance or turn inaccessible due to network errors or shutdowns. Thus, a module has to be distributable across different module repositories from where it can be transparently retrieved and dynamically deployed on the requesting node without the explicit involvement of the user.

Distributed Updates. Another issue is the evolution of applications and their constituent modules. While a missing module can be easily detected by the requesting Internet application system, it is usually not possible to determine whether somewhere a newer variant is available without issuing a query. Hence, collaborating module repositories should be automatically updated after a new variant has been released.

2.2 Composition

Another basic concern is composition which is typically tightly coupled with deployment. While deployment has to deal with the *separation* of an application, composition is about *assembling* an application. It is usually performed during runtime and relies on the deployed modules. The following issues must be considered regarding composition in distributed Internet application systems.

Separation of Deployment and Composition. A major advantage of composable applications is their ability to arrange the real composition after the actual development has been finished. A particular problem of composition within distributed Internet application systems is the diversity and permanent change of component deployment due to the large range of changing system and network constellations. Thus, the composition process should be configurable independent of the current deployment scenario and the hosting Internet node.

Dynamic Composition. While the location of deployed modules on a single Internet node can be easily tracked and resolved, this is not always possible when each module has been deployed on a different Internet node. The related issue is about how modules can be transparently queried and retrieved. Furthermore, after an appropriate module has been located it has to be dynamically retrieved, which may happen in different ways. However, the application should be able to transparently request and access each module in the same way independent of its location.

Multi-Composition System. As mentioned in the introduction, application servers are supposed to host more than one application concurrently. In this scenario, different applications may refer to the same modules. Thus, the composition process has to support *multi-composition*, sharing the same modules and components with several applications. In turn, there are modules which have to be used exclusively by each application, e.g. because of security reasons. Consequently, the composition process should be customizable with respect to sharing and shielding components.

2.3 Related Work

After having highlighted the special concerns of deployment and composition in distributed Internet application systems, in the following we examine various approaches along with related work regarding these issues.

Native Runtime Environments. As mentioned above, deployment is an essential part of application installation and subsequent dynamic composition. For example, Microsoft Windows dynamically deploys components using *Dynamic Link Libraries (DLL)* and UNIX derivates like Linux use *Shared Libraries*, respectively. While they allow encapsulating various components, they have to be packaged in particular files and typically installed in certain paths. Consequently, they require applications to know the physical location of the DLL or the shared library, as illustrated in fig. 1.

```
HINSTANCE hInst = LoadLibrary("c:\sdk\components.dll");
```

Fig. 1. Loading a dynamic link library in MS Windows

Apart from using unique filenames, there is no native versioning support in MS Windows, since the loader distinguishes DLLs by their module names and prevents to load another DLL using the identical module name into the same address space. Moreover, the application can not inspect or query for DLLs matching certain properties without loading them first into memory. And even then, only the basic information can be reviewed, such as module name or particular function entry points. There is no support for revealing the contained components or other resources, simply because DLLs are not primarily intended to carry queryable components but library functions which can be addressed by their well-known name, as shown in fig. 2 for `MyFunc`.

```
typedef bool (*MYFUNC)(void*);
MYFUNC mf = (MYFUNC) GetProcAddress(hInst, "MyFunc");
```

Fig. 2. Retrieving a library function from a MS Windows DLL

Regarding lookup and loading, the native library loader can only retrieve libraries which have been deployed on the local machine. There is typically no support to query remote module repositories for requested libraries. On the other hand, MS Windows DLLs and UNIX shared libraries can be easily deployed by simply copying them into the appropriate directory where the loader is looking for them by file name. As a result, the native development and runtime support for dynamic deployment and composition of components is limited by the constraints mentioned above. Instead, developers have created workarounds like well-known plugin directories to look for components or special configuration files where the location and properties of components can be queried. But these are proprietary approaches and can not be easily ported to other application scenarios.

Virtual Runtime Environments. Besides *native runtime environments* which are inherently bound to certain operating systems like MS Windows or Linux, there are *virtual runtime environments* used in Sun Java or MS .NET [13, 14]. They are particular suited for Internet applications in that they enable the same application executable to be run on different platforms without re-compilation. Regarding dynamic deployment and composition in Java, it differs from the native approach used for C programs in MS Windows and Linux. The fundamental difference is the granularity of loading components. While components of C programs have to be encapsulated in a DLL or a shared library and can only be retrieved together, Java classes can be separately loaded from *Java Archives* provided that the archives and classes are configured in the CLASSPATH, as shown in fig. 3 for the class `mypack.MyClass`.

```
Class c = Class.forName("mypack.MyClass");
mypack.MyClass o = (mypack.MyClass) c.newInstance();
```

Fig. 3. Loading a single class in Java

A variant is the use of a custom class loader which mainly extends the way how classes are located and loaded but not how they are selected, e.g. by given version or vendor properties [15]. Another approach is MS .NET whose dynamic deployment and composition capabilities heavily rely on so called *assemblies*. They represent an improvement over DLLs in that they contain additional metadata and a manifest file which specify further details of the assembly like version number, vendor etc. While the versioning information is evaluated by the *assembly resolver* only for *shared assemblies* stored in a global directory, *private assemblies* are exclusively used by one application and are stored in the path of the application installation directory. Similar to a DLL, an assembly can be easily loaded during runtime, as shown in fig. 4 for `myassembly.dll`. Via the static method `Load` of the class `Assembly` an assembly is dynamically loaded and assigned to the application. The example in fig. 4 also illustrates the use of reflection with `GetTypes` to inspect the content of the assembly.

```
Assembly a = Assembly.Load("myassembly.dll");
Type[] types = a.GetTypes();
```

Fig. 4. Loading an assembly in MS .NET

While virtual runtime environments represent an abstract execution layer on top of the actual platform environment, they still rely on the underlying deployment constraints, such as system environment settings in Java or a well-known directory as in MS .NET. Thus, they are not designed to support distributed deployment and composition scenarios across several and differently configured platforms.

Frameworks. Native and virtual runtime environments are often extended with particular frameworks which add special features regarding dynamic composition, whereas deployment is rarely supported further [16]. Well-known examples with particular composition support are *Sun EJB*, *CORBA Components*, or *Apache Avalon* [8, 17, 18]. They introduce component models which allow composing applications from independently developed components for different purposes but still rely on the inherent deployment scheme of the underlying runtime environment. Thus, regarding distributed Internet application systems, they are supposed to be primarily used on a single platform and are not designed for cross-platform application environments. However, there is also support for special deployment features in selected application scenarios. As an example, web applications can be easily deployed using so called *Web Archives* [19]. The corresponding Java Servlet Engine dynamically configures an appropriate application environment and instantiates the web application. But once deployed, the web application will not be updated if a newer version replaces the former archive. Furthermore, the web archive approach is only applicable when the entire application can be packaged within a single file but not if it is to be composed and completed with other, external components. Another example is *Sun Java Web Start* [6]; it eases the deployment of *Java Applets* in that it organizes downloaded Java archives on the client side in a locally managed cache. Each time an applet is to be started, Java Web Start compares the cached version with the server version and downloads the applet only if there is a newer version on the server. However, it relies on Java archives and thus it inherits the same problems as mentioned above.

Moreover, while it is a good starting point for distributed Internet application systems, it is not able to directly communicate with JAR repositories but by downloading and evaluating a *Java Native Launch Protocol (JNLP)* configuration file from a web server. And it does also not allow a customer to dynamically change the provided configuration of a JNLP application. Finally, there are related frameworks and approaches dealing with composition by refactoring legacy code using *Aspect-Oriented Programming (AOP)*. However, their objectives are different in that they focus on increasing the modularity and configurability of legacy code by using AOP to single out orthogonal features and compose them into aspects [20]. Moreover, due to AOP which is typically applied during compile time, these frameworks are often limited regarding distributed deployment and dynamic composition configuration during runtime.

Application Servers. While a framework typically extends a native application runtime and/or development environment with special features and is usually deployed along with the application itself, this is not feasible for all kinds of extensions. As an example, for concurrently managing *multi-applications* like web applications or web services a so called *application server* is needed. It is started before the actual applications are loaded and then launches each application and service as part of the same process. Related applications are developed using a certain application model and can be solely executed in the target application environment provided by the application server; examples are *Jtrix*, *Apache Avalon*, *Jakarta Tomcat*, *Sun ONE*, *JBOSS* [10, 18, 19, 21, 22]. While the application server approach is feasible for managing multiple applications and an enhanced runtime environment, it is typically still limited to a single platform architecture and a certain application model. Moreover, due to their orientation towards well-defined server-side scenarios with fixed system configurations, application server approaches have not been designed to be configured on the fly for the dynamic composition of new applications or services. Thus, the composition process is limited to components previously installed and known on the target platform and can not dynamically include custom modules provided by remote users on other Internet nodes. An interesting approach in this direction is provided by Jtrix which is not fixed to a single host but targets *code mobility* across multiple platforms. It propagates so called *netlets* which represent a certain kind of Java service. They can be dynamically retrieved from a remote repository and instantiated on the current platform as well as migrated and spread across different nodes. In this sense, Jtrix represents a particular cross-node application server with respect to nomadic services but therefore it only supports service-based deployment and composition, respectively.

In summary, there are particular requirements regarding deployment and composition in distributed Internet application systems. Although there are existing approaches and solutions which address some of them, deployment and composition are basically supposed to be employed in a single Internet application system, in particular application scenarios or fixed system configurations, respectively. Thus, they lack basic support for transparent retrieval of locally and remotely managed components, sharing and shielding of loaded components and distributed synchronization of deployed components.

3 Module Federations

In the following, our approach to deployment and composition using so called *modules* and *module federations* is presented along with its features and its realization in Java and C++.

3.1 Conceptual Approach

According to the requirements of distributed Internet application systems described above, first of all the *logical composition* of application systems should be separated from the *physical deployment* aspects of the involved Internet platforms. For this purpose, we introduce so called *modules* which are special assemblies of components and represent virtual deployable units with well-known resources, unique module ids and property tags. They can be logically retrieved and transparently resolved across different physical deployment scenarios given the module id. Internet applications do not longer work directly with the deployed units of the native approach, e.g. Java archives or MS Windows DLL, but they refer to modules for their composition requests. From this point of view, modules enable the separately configurable *virtual deployment* on top of the *physical deployment*, as illustrated in fig. 5.

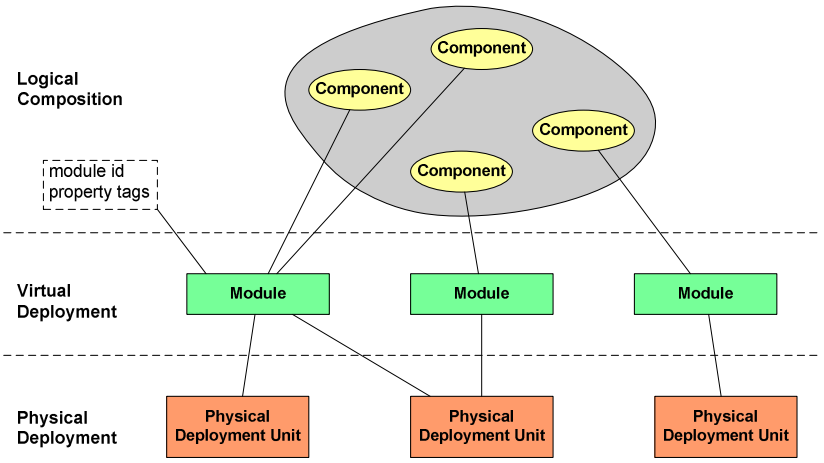

Fig. 5. Physical Deployment, Virtual Deployment and Logical Composition using Modules

A module is acting as a mediator between the physical deployment units and the logical components. It shields the application system and its composition requests from the currently underlying host platform and its configuration scenario. In contrast to native libraries and other deployment units, modules are managed within so called *module repositories* which do not have to be located on the same platform but can be remotely found, queried and managed, as shown in fig 6. A *module controller* on platform AS handles the module loading requests received from the hosted

applications and transparently retrieves the module from a possible remote module repository. This is particularly suitable for distributed and decentrally organized system configurations like peer-to-peer networks or mobile scenarios.

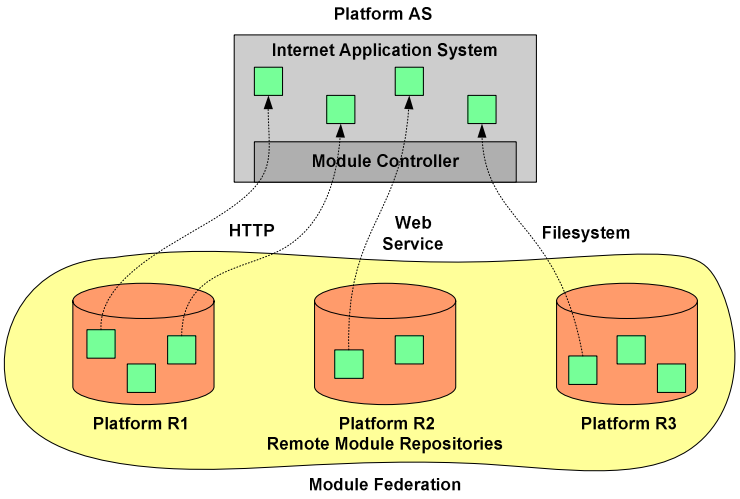

Fig. 6. Module repositories and module federation

Moreover, module repositories can be organized in so called *module federations* for sharing and synchronizing modules across distributed Internet platforms, as also shown in fig. 6. While there are various options to manage a federation, its nodes and the distributed resources [23], our approach does neither rely on a particular organization nor communication protocol as long as there is an appropriate plugin to enable the module controller to interact with the federation or a single repository and shield the logical application composition from the physical module deployment. As an example, the discovery of available module repositories could be managed by manually edited local configuration files, well-known directory services or peer-to-peer approaches. Some repositories may provide network access to their modules over HTTP whereas others may use web services and SOAP. Furthermore, the localization of a module can be performed by querying each module repository one-by-one or by a lookup in a central directory service based on LDAP where each module has been registered previously. In effect, the federation hides the physical deployment of modules across various remote nodes, and once a module has been deployed into the federation, each Internet application system can transparently query and retrieve this module. From this point of view, a module federation represents a group of well-known, trusted and collaborative module repositories which finally behave like a single virtual module repository. Due to the introduction of modules which act as mediators between *deployment constraints* of the involved platforms and *composition constraints* of the concerned application systems, the approach is especially suitable for flexible application systems and variable platform scenarios. It is not bound to a certain programming language feature or operating system and is open to package

different kinds of components like classes, binary resources or programming libraries and to dynamically deploy them into module repositories by request.

3.2 Features

The main features and benefits of the proposed conceptual approach regarding deployment and composition in distributed Internet application systems are as follows.

Custom Component Packaging. An important issue concerning component deployment is the custom packaging into deployable units. While the native deployment items like DLLs or Java archives originally lack support for registering, describing and retrieving single components, a module provides options to arbitrarily manage and assemble components within a deployment unit.

Queryable Description. While deployment units are typically addressed using absolute filenames or well-known identifiers, there are often different variants which can only be selected by evaluating custom properties like e.g. versioning information. Thus, remote module repositories can be queried for appropriate modules and components without actually downloading the module.

Transparent Handling of Variants and Dependencies. A component is often used in conjunction with other components. The result is a dependency graph between components and deployment units, often across different variants. Our approach hides the dependency handling from the application in that it provides a uniform way to retrieve modules and contained resources without bothering the developer to manually resolve possible module dependencies.

Distributed Module Repositories. A frequent constraint of current Internet application systems is their limitation to be composable only of locally deployed components. Our approach uses module repositories which can be locally or remotely found and supports the transparent sharing of a deployed module. Thus, a developer does not have to update every involved Internet application system but must only deploy the module once into a single module repository.

Dynamic Deployment and Composition. In native application scenarios, deployment units like Java archives or MS Windows DLLs have to be deployed by the site administrator before the application can be started. In contrast, our modules can be dynamically deployed into a module repository and retrieved by each application in the module federation without customizing the current host platform.

Module Handler. Native deployment units have only limited support for managing the unit during runtime. For example, there is basically no central initialization of the Java archive when a contained Java class or component is accessed the first time. Our

approach comes with a module handler which provides a uniform API for accessing the contained resources and tracks their usage.

Shared and Shielded Module Instances. A single application system can easily track down which modules have already been loaded and will not load the same module more than once. On the other hand, Internet application systems are often dealing with concurrently loaded services. We ensure that shared modules are only loaded and instantiated once. Subsequent requests return the same module instance and allow the reuse of resources across concurrently hosted applications.

3.3 Realization

As mentioned above, our conceptual approach is neither bound to a certain programming language feature nor operating system. Thus, in the following we describe the realization of the approach in an exemplary fashion for Java and C++ for MS Windows. We will primarily focus on how to work with modules regarding development, deployment, composition and configuration issues. The tasks of how to implement and synchronize a federation are not covered in detail since they have been already addressed in other works [23]. In fact, the interaction with the module federation and the participating nodes is actually performed by module controller plugins which hide the details of discovering module repositories, querying modules and downloading them onto the requesting Internet node.

Java Implementation. In contrast to MS Windows or Linux runtime environments, the Java Virtual Machine (JVM) does not couple a physical deployable unit one-by-one with logical composable entities. Each Java class within a deployed Java archive can be independently retrieved and used for composition without addressing the other classes in the same archive. However, this is not valid for MS Windows DLLs or shared libraries of Linux which have to be completely loaded for retrieving a contained component. Thus, the basic question for realizing the approach in Java is how to define the required classes of a Java module when there is no option to group classes logically but only physically. We have addressed this problem in previous work by introducing so called *class collections* [24], as shown in fig. 7.

```
<collection name="sun-jaf" id="sun-jaf">
 <variant> <property name="release" value="1.0.1"/>
  <file location="/sdk/sun-jaf-1.0.1/activation.jar">
    <package name="com/sun/activation/.*"/>
    <package name="javax/activation/.*"/> </file>
 </variant> </collection>
```

Fig. 7. Grouping Java classes using class collections

Each collection definition contains a unique id that can be used to refer to this collection like *sun-jaf* in fig. 7. The subsection can then define different variants of the collection with various properties which are later also used to select a particular variant among several variants. Finally, the location of the JAR file where the

contained classes can be found is given in conjunction with class name patterns that specify which classes can be loaded from the JAR files. To evaluate the collection configuration, the system class loader of the JVM is replaced by a custom class loader which checks each class loading request and determines the right class according to the configuration file. The module configuration in turn uses class collections to specify the required classes when the module is about to be used, as shown in fig. 8.

```
<module name="texteditor" id="{A9D52EF1}"
           handler="de.fraunhofer.texteditor.CModule">
 <property property name="vendor" value="Fraunhofer" />
 <dependency>
   <module id="{2E6210AA}"/><module id="{D181334A}"/>
 </dependency> <collection id="texteditor">
   <property name="release" value="1.0.0" />
 </collection> <collection id="apache-xerces">
   <property name="release" value="2.4.0" />
 </collection> </module>
```

Fig. 8. Module deployment using class collections

The module `texteditor` is marked with a *globally unique identifier (GUID)* and defined to use a collection `texteditor` and `apache-xerces` with the given properties. As a result, the module configuration does not longer rely on Java archives or has to specify exactly which classes are needed. Instead, the class collections shield the composition of the module from the actual deployment of the required classes. The attribute `handler` points to a class which represents the module handler of the current module, e.g. performing the initialization or providing access to its resources. Furthermore, the module may also define properties like `vendor` which can be used to query this module. Finally, the `dependency` section indicates which modules have to be loaded by the module controller before the current module can be used.

After the physical deployment of modules, we have to define the logical composition, configuration and management of loaded modules. What will happen when the same module is requested in two different variants by two applications hosted within the same JVM? Or how can an application determine whether the module to be retrieved is already loaded and initialized or not? The first question is targeting a basic problem of the original class loader approach of Java which does not allow loading two classes having the same *fully-qualified class name (FQCN)* by the same class loader [25]. Thus, in order to support the composition of modules in different variants within the same JVM, we have to use and manage several class loaders. This problem has been also addressed by our previous work introducing so called c*lass spaces* [26]. They enable developers and administrators of Internet application systems to configure exactly which Java classes and class collections are shared across or shielded from other concurrently loaded applications. The class space approach is used as the basis to specify which modules are shared and which are shielded by introducing so called *module spaces* as shown in fig 9. There is one module space `shared` and two child module spaces `shielded-1` and `shielded-2`. While the module space `shared` is configured to load two modules

which are shared across `shared`, `shielded-1` and `shielded-2`, the latter two are organized to hold only one module which is not seen by any other module space.

```
<modulespaces> <space id="shared" parent="application">
   <module id="{36242453}"/> <module id="{BE441538}"/>
  </space>
  <space id="shielded-1" parent="shared">
   <module id="{A9D52EF1}"/>
     <property name="vendor" value="FhG" /> </module>
  </space>
  <space id="shielded-2" parent="shared">
   <module id="{2EBF97FD}" /> </space>
 </modulespaces/>
```

Fig. 9. Module sharing and shielding using module spaces

In case a shared module is to be requested a second time, it is not loaded again but its reference is returned to the caller without initializing the module twice. In general, the module spaces are organized in a hierarchical structure where modules in a child module space can only share the modules on the path to the root space. In turn, the modules located in other module spaces are shielded. In effect, our module management completely hides the issues of lookup, loading and initializing modules from the application, as depicted in fig 10. The application can simply request a module by specifying the related module id `modId1` and will get a reference to it.

```
CModuleId modId1 = new CModuleId("{657B3CA5 }");
IModule mod1 = getModuleManager().openModule(modId1);
```

Fig. 10. Requesting a module in Java

The module requestor can also pass additional parameters describing the desired module like a property list, or the related module space is accordingly configured with properties like e.g. `vendor`, as already shown in fig. 9. The module resolver plugin evaluates these additional parameters before loading the module and tries to resolve an appropriate module. However, there are also other resolving approaches, such as *semantic trading* as used in [27], which evaluate particular aspects like module behavior and the current composition context. For this purpose, additional resolver plugins can be included in our approach that model the related resolving scheme.

After the module has been successfully loaded, it will be initialized using the method `init` within the module handler object. This method is called only once and can be used to setup the module, e.g. register contained resources or request other required modules like `mod2` as shown in fig. 11. In turn, when the last user of the module has released its reference, a corresponding method `exit` is called which can be used to release acquired modules or to cleanup other resources.

```
public void init() {
  registerResource("{B5C91A0B}", new CMyResource());
  IModule mod2 = openModule("{CDACCCD7}"); }
```

Fig. 11. Initializing a module

The next step for the module requestor is to access the contained resources and components. For this purpose, each module exposes a particular module handler interface which can be used to query a module for well-known resources by unique identifiers. As already described above, we do not want to address the dynamic negotiation, adaptation and gluing of arbitrary components, e.g. by using contracts to describe the interface semantics [28]. Rather, we want to support the transparent deployment and composition of already collaborative and suitable components which are distributed and managed on remote Internet platforms. Thus, if a requested module has been found and loaded, there is no longer the question whether and how the contained components will fit but only how to get a reference and to access them, as shown in fig. 12. A component within the module is requested by `openResource` given the corresponding resource id which has also been previously used to register the resource and as depicted above.

```
IResource res = mod1.openResource("{B5C91A0B}");
```

Fig. 12. Accessing a certain resource within a module

Finally, the module approach has been implemented in Java in conjunction with class collections and class spaces [26]. It offers developers of components, configurators of applications and administrators of Internet platforms a unique way to separately define physical deployment, logical composition and configuration of Java components. Along with the distributed management of modules within module federations it represents a transparent foundation of composable distributed Internet application systems written in Java.

C++ Implementation. Applications compiled into platform-dependent executables as in the case of C++ programs can not be directly transferred to arbitrary platforms like with Java. At least the source code must be compiled for different platform architectures, although the basic programming strategy may remain the same. Therefore, while we limit the discussion in the following to the realization with C++ and MS Windows, our approach has also been implemented for Linux environments in a similar way. In MS Windows, the deployment strategy of dynamically loadable components mainly relies on DLLs which can be independently deployed and easily incorporated into an application. Basically, there are two ways. The first automatically retrieves the DLL before an application is loaded. If the DLL can not be located and loaded, the application is not started. Using DLLs in this way is the simplest option for developers because there is no need to change the program code compared to linking an application against static libraries. However, the administrator of the hosting Internet platform or the application itself can not adjust the resolution strategy or loading process. The required DLL must be in the path of the application, otherwise the DLL loader will not be able to find the DLL. The second way is more dynamic. The application itself can request a certain DLL during runtime and dynamically specify an arbitrary file location as long as the DLL is stored somewhere on a mounted file system. In contrast to a Java archive, a DLL represents more or less a closed deployment unit. It does not allow inspecting or retrieving parts of it as in the case of loading a single Java class out of a Java archive. Everything contained in the DLL will be always deployed and retrieved completely in an all or nothing fashion.

However, this also greatly supports the packaging of a module within a DLL. As an example, fig. 13 shows a configuration file used by a module repository which defines a module that is tagged with the property `vendor`. There is no need for an additional class collection configuration file as in the case of Java archives. But similarly, instead of relying on the physical deployment unit DLL and its possible varying location on different platforms, an application can virtually request modules by issuing an `openModule` to the module controller with the related module id. The module controller will then load the configured DLL and initialize the contained module similar to the Java implementation showed above.

```
<module id="{A9D52EF1}">
 <property name="vendor" value="Fraunhofer"/>
 <dll id="teditor" loc="http://crossware.org/teditor"/>
</module>
```

Fig. 13. Module deployment using MS Windows DLLs

In addition, we again use *module spaces* like in the Java implementation and are able to define shared and shielded modules similar to the discussion above. After all, a module has to be initialized when it is loaded the first time. In contrast to Java, there are several ways how modules can be used in MS Windows applications. They may be part of a static library linked with the application, they are implicitly loaded with a DLL when the application is started or they are explicitly loaded by an application on request. Either way, an Internet application system with different loaded applications has to ensure that each module is only loaded and initialized once. But in contrast to Java, there is a problem referencing a particular class out of a DLL. There is no way to get access to single classes but only to exported functions, as described above in fig. 2. However, the module management must know which modules are contained within the DLL and how to get a reference to them. To solve this problem, each module is automatically registered to the module controller when the DLL is loaded. For static libraries, static methods are used to do that when the application is started. For implicitly and explicitly loaded DLLs, the corresponding DLL initialization function `DLLMain` is used to register all contained modules, as shown in fig. 14.

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD
fdwReason, LPVOID lpvReserved){
 m1=GetModuleManager()->RegisterModule(&Tools::Module);
 m2=GetModuleManager()->RegisterModule(&Edit::Module);}
```

Fig. 14. Registering of modules packaged within the same DLL

In effect, each module is registered and only initialized once in a similar way to the Java realization. The differences between modules contained in static libraries, implicitly or explicitly loaded DLLs are hidden from the caller and therefore a common API can be used to request a module, as shown in fig. 15.

```
CModuleId modId1 = new CModuleId("{AA9C391B}");
IModule mod1 = GetModuleManager ()->OpenModule(modId1);
```

Fig. 15. Requesting a module in C++

After getting a reference to the module controller, the desired module is requested by `OpenModule` passing the related module id `modId1` or eventually further parameters which may be evaluated by certain module resolver plugins as already described for the Java implementation. Whenever the module controller is called to retrieve a module, it transparently loads and initializes the module and returns a reference to it. In case the module has been previously loaded, it directly returns the related object reference. This way, if a module originally requested by the application needs also other modules, they are resolved without explicit intervention of the application. The module manager also pays attention that the same module is not loaded twice within the same module space and that it is not initialized repeatedly.

3.4 Discussion

The presented proposal has been implemented using Java and C++ with MS Windows and Linux, having had in mind to introduce a common, *platform- and programming language independent approach* to distributed software deployment and dynamic application composition. As a result, the actual underlying deployment and composition strategy is completely hidden and its configuration is separated. Modules containing components and other deployed resources can be transparently retrieved and are automatically instantiated. In contrast to native, deployment dependent composition approaches like MS Windows DLLs or Java archives, the developer can focus on the business logic and dynamically request a certain component or resource without considering how and where the resources have been actually packaged and deployed. In effect, the tasks of composition and deployment are cleanly separated among developers and deployers of modules and administrators of an Internet application system, respectively. With respect to existing source code, the approach can be seamlessly added and used without introducing a particular packaging or deployment strategy as in the case of Java servlets or a heavy-weight composition framework like Sun Enterprise Java Beans (EJB). Instead, it is built upon native packaging approaches like Java archives and MS Windows DLLs in conjunction with particular module loaders and configuration files that can be extended to support different resolving schemes. In addition, it requires only small changes of the source code of how components are retrieved and accessed. Furthermore, the presented approach allows customers to logically compose an application on their own and is not limited to existing native deployment units like Java archives and predefined composition configuration as in the case of JNLP-based approaches. Another important point is runtime performance. In comparison to native approaches like the original Java classloader or DLL loader of MS Windows, the realizations in Java and C++ reveal overhead only for the localization of an appropriate module matching the module request and loading it from a possibly remote module repository. This depends heavily on the actual application scenario and represents the cost for a highly configurable and distributed component deployment. However, the loading time can be reduced by installing a local module repository which is caching retrieved modules or is automatically synchronized by the module federation. Finally, the actual access on a component within an already loaded module and subsequent composition remain as fast as with the original approach.

4 Application of the Approach

In the following, we depict the application of our approach on *cross-platform computing* [29] where several Internet nodes are grouped to create a multi-platform application environment. Each node is capable to host an arbitrary application which in turn is dynamically composed of components deployed by various developers within a so called *platform federation*. We use this concept in the ongoing research project *CAT* [30] for the development of the Java-based open community platform *netzspannung.org*. Each member of the platform is encouraged to develop new components for the system and to offer them to other members by deploying the related modules into provided distributed module repositories as shown in fig. 16.

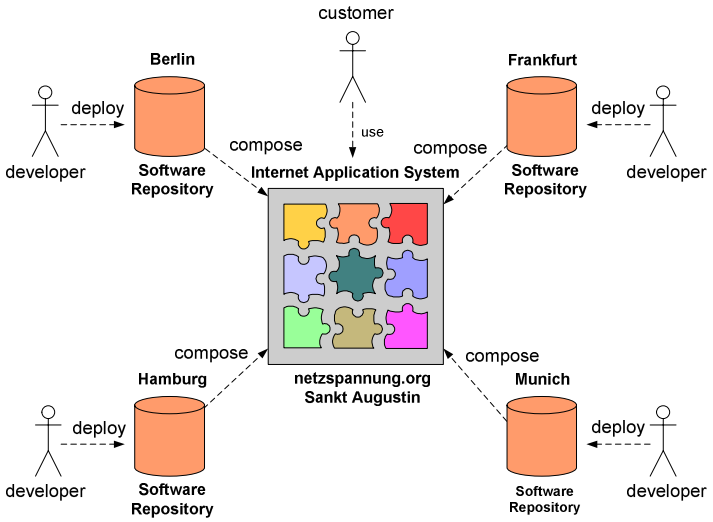

Fig. 16. Dynamic deployment and composition in netzspannung.org

A developer who wants to add a new component to the system first has to add a module handler to his/her project and implement the `init` method as described in section 3.3. There, (s)he registers and initializes all components, resources and objects to be accessible using the module handler. Next, after having packaged all related Java classes in JAR files, (s)he runs a provided tool `collection` to automatically create a collection configuration file of the related JAR files indicating the concrete classes that can be found in there. In this file (s)he also adds configuration lines to specify which third-party Java classes are also needed. Then, (s)he manually creates a module configuration file listing the class of the module handler, constraints of the module and the required class collections, as shown in fig. 8. Finally, the developer uses a helper tool `mddeploy` to upload the JAR files, the collection and module configuration files to a participating module repository. As a result, the configuration files can be used to inspect the module as well as the constraints of the underlying JAR files without actually downloading them from the repository, as described in

section 3.3 and in [26]. In case a module represents a new application to be published on netzspannung.org, it must be registered with its module id in the application list of the site configuration by the administrator. Consequently, when the application is to be started, the corresponding module id is taken from the application list and the platform module controller is querying the module repositories for the related module. It automatically resolves the specified dependencies, downloads the module code and uses the information in the module configuration to find the class of the module handler for initialization. Finally, the module is initialized and in turn can request further required modules. This way, each module is loaded one-by-one and the application finally gets composed without knowing from where and how the modules are retrieved. In effect, a customer is able to start each application on the platform and gets the illusion that everything has been deployed on a single host instead of different nodes. Based on the *netzspannung.org* platform (which is operational since about two years) and its module repositories we are currently conducting a further research project called AWAKE [31] which deals with knowledge management via the Internet. Besides particular server-side components, an important part is the client-side user interface for accessing the knowledge space of AWAKE and working with different views. For that, we developed a so called *Internet Application Workbench* which provides a desktop-like GUI and works with modules retrieved from remote module repositories similar to the *netzspannung.org* platform. However, the underlying module framework is not pre-installed but dynamically established on the client-side using Sun Java Web Start. But in contrast to pure JNLP-based approaches like *Object Component Desktop* [32], the actual subsequent composition of an application is not performed using a JNLP file, but can be individually customized by the customer choosing modules from different providers and module repositories.

5 Conclusions

In this paper, we have investigated distributed Internet application systems with respect to their functional requirements to deployment and composition. We have argued that existing solutions lack transparent support for distributed deployment and dynamic composition of remotely located and managed components. Consequently, we have presented a novel solution for solving these problems by introducing so called modules and module federations. Their realizations in Java and C++ for MS Windows were described, and the suitability of our proposal was demonstrated by presenting its application in ongoing research projects. As a result of our approach, different deployment scenarios are hidden from the application and can be easily tackled during runtime using particular composition and deployment configuration files. In effect, the Internet is turned into a distributed Internet application system where each node can be equally used to host applications which are dynamically composed of remotely managed components. Although we are already using the approach in different projects and public installations like netzspannung.org, we are still investigating how to extend existing and include new features. A basic problem of the current deployment strategy is the lack of authentication when a new member is registering to become part of a module federation. Currently, each module repository

must be configured to trust the new module repository and its managed modules. The same is valid for a new developer who wants to deploy a module on a new module repository. Another issue concerns access and composition control and whether a particular component can be used or combined with other components in a certain Internet application system. In this context, the presented approach could be also employed to open a new business model for software leasing, which could be called *feature leasing* of composable applications. Depending on the leasing contract, a customer may be able to use only selected features, and the application is only composed with the requested functionalities. Moreover, thinking of client side applications whose components have been downloaded over the Internet, updated components may be only retrieved and installed after upgrading the leasing contract. The resolution of modules is currently performed by resolver plugins matching property lists. However, we are already working on the implementation of plugins for more flexible semantic-based module trading. Finally, the concept of the presented approach is not limited to Java or Linux. Basically, it could also be used to realize a similar module strategy for MS .NET. However, as long as MS .NET is not available for a platform type different than MS Windows, its application is limited and typically not feasible in a heterogeneous network environment like the Internet.

6 Acknowledgements

The presented approach has been applied in the ongoing research projects CAT and AWAKE which are financially supported by the German Ministry of Education and Research (BMBF). The projects are conducted by the research group MARS of the Fraunhofer Institute for Media Communication, Sankt Augustin in cooperation with the University of Siegen and the University of Marburg, Germany. Special thanks go to Monika Fleischmann, Wolfgang Strauss, Jasminko Novak and Daniel Pfuhl.

References

1. Schatz, B. R. The Interspace: Concept Navigation Across Distributed Communities. IEEE Computer. Vol. 35, Nr. 1. pp. 54-62. IEEE 2002.
2. Vaughan-Nichols, S. J. Web Services: Beyond the Hype. IEEE Computer. Vol. 6, Nr. 2. pp. 18-21. IEEE 2002.
3. Vinoski, S. Web Services Interaction Models - Putting the “Web” into Web Services. IEEE Internet Computing. Vol. 6, Nr. 4. pp. 90-92. IEEE 2002.
4. Milenkovic, M., Robinson, S. H., Knauerhase, R. C., Barkai, D., Garg, S., Tewari, V., Anderson, T. A., Bouwman, M. Toward Internet Distributed Computing. IEEE Computer. Vol. 7, Nr. 5. pp. 38-46. IEEE 2003.
5. Lawton, G. Distributed Net Applications Create Virtual Supercomputers. IEEE Computer. Vol. 33, Nr. 6. pp. 16-20. IEEE 2000.
6. Srinivas, R. N. Java Web Start to the Rescue. JavaWorld. IDG 2001. Nr. 7. http://www.javaworld.com/javaworld/jw-07-2001/jw-0706-webstart_p.html
7. Netx. http://jnlp.sourceforge.net/netx
8. Monson-Haefel, R. Enterprise Java Beans. O'Reilly & Associates. 2000.

9. Vinoski, S. Web Services Interaction Models - Current Practice. Internet Computing. Vol. 6, Nr. 3. pp. 89-91. IEEE 2002.
10. Silver, N. Jtrix: Web Services beyond SOAP. JavaWorld. IDG 2002. Nr. 5. http://www.javaworld.com/javaworld/jw-05-2002/jw-0503-jtrix_p.html
11. Fugetta, A. Picco, G. P., Vigna, G. Understanding Code Mobility. IEEE Transactions on Software Engineering. Vol. 24, Nr. 5. pp. 342-361. IEEE 1998.
12. Ning, J. Q. Component-Based Software Engineering (CBSE). Proc. of the 5th Intl. Symposium on Assessment of Software Tools (SAST). pp. 34-43. IEEE 1997.
13. Eckel, B. Thinking in Java. Prentice Hall. 2002.
14. Prosise, J. Programming Microsoft .NET. Microsoft Press. 2002.
15. Gong, L. Secure Java Class Loading. IEEE Internet Computing, Vol. 2, Nr. 6. pp. 56-61. IEEE 1998.
16. Fayad, M. E., Schmidt, D. C., Johnson, R. E. Implementing Application Frameworks: Object-Oriented Frameworks at Work. John Wiley & Sons. 1999.
17. Marvic, R., Merle, P., Geib, J.-M. Towards a Dynamic CORBA Component Platform. Proc. of 2nd International Symposium on Distributed Objects and Applications (DOA). Antwerpen, Belgium. pp. 305-314. IEEE 2000.
18. Apache Server Framework Avalon. http://jakarta.apache.org/avalon/framework/index.html
19. Goodwill, J. Apache Jakarta Tomcat. APress. 2001.
20. Zhang, C., Jacobsen, H.-A. Quantifying Aspects in Middleware Platforms. Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD). pp. 130-139. ACM 2003.
21. Watson, M. Sun One Services (Professional Middleware). Hungry Minds. 2002.
22. JBOSS Application Server. http://www.jboss.org
23. Lestideau, V., Belkhatir, N., Cunin, P.-Y. Towards Automated Software Component Configuration and Deployment. Proc. of the 8th Intl. Conference on Information Systems Analysis and Synthesis. IIIS 2002.
24. Paal, S., Kammüller, R., Freisleben, B. Java Class Deployment with Class Collections. Proc. of the 3rd International Conference for Objects, Components, Architectures, Services and Applications for a Networked World (NODE). Erfurt, Germany. pp. 144-158. 2002.
25. Liang, S., Bracha, G. Dynamic Class Loading In The Java Virtual Machine. Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 36-44. Canada 1998.
26. Paal, S., Kammüller, R., Freisleben, B. Customizable Deployment, Composition and Hosting of Distributed Java Applications. Proc. of 4th Intl. Symposium on Distributed Objects and Applications (DOA). LNCS 2519. Irvine, USA. pp. 845-865. Springer 2002.
27. Bernard, G., Kebbal, D. Component Search Service and Deployment of Distributed Applications. Proc. of 3rd Intl. Symposium on Distributed Objects and Applications (DOA). pp. 125-135. IEEE 2001.
28. Crnkovic, I., Hnich, B., Jonsson, T., Kiziltan, Z. Specification, Implementation, and Deployment of Components. Communications of the ACM. Vol. 45, Nr. 10. pp. 35-40. ACM 2002.
29. Cusumano, M. A., Yoffie, D. B. What Netscape learned from Cross-Platform Software Development. Communications of the ACM. Vol. 42, Nr. 10. pp. 72-78. ACM 1999.
30. Fleischmann, M., Strauss, W., Novak, J., Paal, S., Müller, B., Blome, G., Peranovic, P., Seibert, C., Schneider, M. netzspannung.org - An Internet Media Lab for Knowledge Discovery in Mixed Realities. In Proc. of 1st Conference on Artistic, Cultural and Scientific Aspects of Experimental Media Spaces (CAST01). St. Augustin, Germany. pp. 121-129. Fraunhofer 2001.
31. AWAKE - Networked Awareness for Knowledge Discovery. Fraunhofer Institute for Media Communication. St. Augustin, Germany. 2003. http://awake.imk.fraunhofer.de
32. Object Component Desktop. http://ocd.sourceforge.net