Rüdiger Kapitza, Franz Hauck

AspectIX: DLS: a CORBA Service for Dynamic Loading of Code

Technical Report TR-I4-02-06
2002-12-16

Friedrich-Alexander-University Erlangen-Nürnberg, Germany

Informatik 4 (Distributed Systems and Operating Systems)
Prof. Dr. Fridolin Hofmann

DLS: a CORBA Service for Dynamic Loading of Code

Rüdiger Kapitza[1], Franz J. Hauck[2]

[1] Dept. of Comp. Science, Informatik 4, University of Erlangen-Nürnberg, Germany
rrkapitz@cs.fau.de
[2] Distributed Systems Lab, University of Ulm, Germany
hauck@informatik.uni-ulm.de

Abstract. Dynamic loading of code is needed when rarely used code should be loaded on demand or when the code to be loaded is not known in advance. The latter is often the case in distributed systems as newly developed code should be executed by running (legacy) applications. At the extreme, distributed programming concepts as mobile agents and intelligent proxies rely heavily on dynamic code loading. Some programming languages support dynamic code loading (e.g., Java) whereas others (e.g., C++) need support by the operating system (e.g., for loading and executing shared libraries). We focus on CORBA an object-based, language-independent middleware architecture. Currently CORBA does not support code loading. Therefore, we specify a CORBA Dynamic Loading Service (DLS) that allows for transparently loading of code modules into a local CORBA environment. Our DLS specification is entirely language independent. We also present an implementation of DLS which is able to identify the code implementation that fits best for the current environment. The selection is not only based on programming language and processor architecture but also on versions of available libraries and on locally executed compatibility tests.

1 Introduction

Applications will need to dynamically load additional code at run-time if that code is not already bound to the local execution environment. Code modules may be rarely used and the application did not want to load it at start-up time (e.g., the plug-in modules of a Web browser). Other code modules may not be known at compile- or even at start-up-time. This is often the case for distributed applications that have numerous, independently running application parts. Newly developed code should be executed by already running execution environments. Additionally, for some wide-area distributed applications it is simply not feasible to install and load all code modules at every node of the system as they will only be used by a few of the nodes. As code usage often depends on the users' interaction with the distributed application those few nodes may not be known in advance.

In this paper, we focus on CORBA an object-based middleware standard for distributed applications [10]. In CORBA, applications are built from distributed objects which can be developed in a variety of programming languages. A so-called language mapping takes care of interoperability between distributed objects written in different languages. CORBA objects can transparently interact with each other across different address spaces and execution environments. CORBA provides distribution transparency—i.e. there is no distinction between the interaction of local and remote objects—and location transparency—i.e. it is not necessary to know the object's location for a successful interaction. The implementation of transparent object interaction is based on a standardized protocol (IIOP, Internet inter-orb protocol), a standardized representation of object references (IOR, interoperable object reference), and the automatic code generation of proxy objects (stubs) and server side code (skeletons) from a language-independent description of object interfaces.

With respect to dynamic loading of code, CORBA does not provide any support. An application may interact with CORBA objects that were not known at start-up time. Only the object reference to such objects has to be handed over to or retrieved by that application. However, there is no support for loading of code that was not known at start-up time or is rarely used. Application programmers have to rely on mechanisms provided by the underlying language used to develop CORBA objects. As every language has more or less support for dynamic code loading the mechanisms to be used are rather different. The language-independent programming model of CORBA is compromised at that respect. Thus, CORBA applications using dynamic code loading are hardly portable. Even worse, modern programming paradigms as mobile agents, mobile objects and intelligent proxies require that previously unknown code has to be loaded into the local environment in order to host an agent or an object, or to bind to special objects. Unfortunately the loading mechanisms are different for every language that may be used. Even for the same language there may be different mechanisms due to differences in the underlying operating systems.

We present DLS, a CORBA service for dynamic loading of code. Like other CORBA services it appears as a pseudo CORBA object that has a well-defined interface. This interface can be mapped to any programming language supported by CORBA. The language-specific code-loading mechanisms finally used by the service are hidden from its users. Thus, DLS can also be used as a generic base mechanism for implementing mobile CORBA objects, mobile agents and intelligent proxies.

Beside the service definition, we will outline an implementation of DLS. This implementation does not only use the code loading mechanisms of a specific programming language, but also identifies the code implementation that fits best for the running environment. The code selection does not only rely on language, processor architecture and operating system but also takes locally executed tests into account. For certain languages and environments it is possible to load code written in one of a variety of programming languages (e.g., in Java we can load Java code as well as native code written in C or C++). Code written in different languages may expose different properties (e.g., efficiency, memory usage, precision to name a few) that can be taken into account for code selection.

Security issues are beyond the scope of this paper. Dynamic loading of code always involves security considerations, and we assume that standard security mechanisms as code signing and a public key infrastructure can be used for securing DLS. However, our implementation does not yet make direct use of such techniques.

This paper is organized as follows: Section 2 will discuss code loading in general. In Section 3, we present DLS as a generic and language-independent CORBA service. Section 4 introduces our current implementation of DLS based on and integrated into the *JacORB* open source CORBA implementation [2]. Section 5 briefly describes how

DLS can be used to implement mobile objects, mobile agents, and intelligent proxies. In Section 6, DLS is compared to related work. Finally, Section 7 gives our conclusions.

2 Dynamic Loading of Code

Almost every modern programming language provides mechanisms to dynamically load and execute code on demand. Languages like Java and Perl have built-in support for dynamic code loading. Other languages like C and C++ can only load code with support by the underlying operating system, which provide mechanisms like dynamic link loaders and shared libraries. Languages with built-in support rely on a naming scheme for loadable modules.

Java loads code on the class level. A compiled Java class is represented in a standardized format and can be loaded on any platform by any Java Virtual Machine. The hierarchical class name identifies the code to be loaded. The class name is usually made unique by integrating an Internet domain name. With the concept of class loaders, Java also supports remote loading of code including security measures as code-signature verification and trusted code sources. Perl loads code in source form and compiles it at runtime. A naming convention for Perl modules identifies the source file in well-defined directories.

C and C++ programs need to use system calls to the operating system to load and bind new code. Those calls depend of course on the type of operating system used. Also C and C++ code is usually compiled for a special processor architecture, for an operating system and, due to alignment and calling conventions, even for a specific compiler family. Thus, the loaded code does not only depend on the programming language but also on the environment it is going to be executed. The loaded code typically has to be accessible as a shared library file in the file system and the loading program has to know the corresponding file name. Java and Perl can also load compiled C and C++ code (so-called native code) in form of shared libraries. These languages provide corresponding operations to access the loading mechanisms of the operating system. Naming conventions take care that finally the file name of a shared library can be determined.

To summarize, the code loading facilities of programming languages can be used to load code on demand in order to load code only when needed. Loading of unknown code that may be even dynamically loaded from the network is not directly supported by most languages. The code first has to be downloaded into a local file and second can be loaded into the local execution environment. Java is an exception as it can transparently load code from other hosts using specialized so-called class loaders.

By viewing the loading of code from the perspective of the loading program we can identify two essential requirements: the dynamically loaded code should have a certain functionality, and it should be executable in the local environment. The latter is entirely neglected in current languages as it is assumed that the code is executable and prepared to run on the designated platform. This is entirely different if we move to CORBA or distributed systems in general. Even if we anticipate a certain programming language it will not be clear whether a certain shared library file will do its job on the local platform except it was exactly built for it. This is especially true if the code was loaded from the network. If a CORBA application is distributed over multiple hosts and furthermore assumed that all parts are written in the same language, it may be necessary to provide different shared library files for the same functionality for the different execution environments used (e.g., Windows XP and Linux). If a CORBA application is written in different languages it may be necessary to provide different code packages written in different languages such that the same functionality can be dynamically loaded into the different execution environments.

This scenario implies that naming of code is handled on the level of functionality and not on the level of file names. Assuming, we have a naming scheme for functionalities, the mapping of a name to a suitable implementation of a specific code functionality depends on a set of criteria:

- instruction set (of a certain processor architecture or interpreter)
- operating system (assumed semantics of calls into the operating systems)
- compiler family (alignment and calling conventions)
- code format (how is the code written to file)

Of course some of the criteria may collapse, e.g., for Java there is currently only one possible format, the Java class file. However, the selection of implementations may be more complicated: The loaded code may depend on other code already installed or to be loaded into the local environment. It may even depend on a certain version of that other code. This is also true for a distributed system of nodes with same processor, same operating system and using the same programming language. The code to be loaded may rely on additional properties of the execution environment. As an example consider a Java program that may want to load a class for fast graphical rendering. The same functionality may be provided by three different loadable code packages:

- a native library written in C++ and assembler using MMX processor instructions.
- a native library written in C++ and
- a class written entirely in Java

Ideally, the system should test whether it is possible to load the first variant as it will be the fastest. If this cannot be done, e.g., the local processor does not support MMX instructions or as the code is not available for the current operating system, the program will try to load the second alternative that is less fast but never the less quite efficient. If this fails too, the third alternative will be loaded. As this implementation is written in Java it will work everywhere.

It is very hard to program this loading strategy into a local application as it requires explicit knowledge about all the alternatives and about their requirements. It is impossible to integrate at all if the code and its alternative implementations are not known at compile time. In fact, we would like to have some service that is asked to load code with a certain functionality. This service is searching for the best implementation of the required functionality and loads it into the local execution environment.

3 Dynamic Loading Service

We present now DLS, a CORBA service for dynamic loading of code. The service's task is to load code on request. Code is referred to by a symbolic name. The symbolic name does not include any information about availability of code for a specific platform, compiler or language. Instead the symbolic name just identifies the semantics of the code's functionality which in turn may be implemented in different variants, e.g., for certain platforms, compilers and languages.

The symbolic name is represented by a simple string. Our service definition does not imply any structure of the name. However, we suggest to adopt the same conventions as used for Java class names: The string should be split into syllables separated by a dot. The first syllables should represent a fully qualified Internet domain name in reverse order. In fact, the symbolic names can be equivalent to Java class names. However, we have to note that this is not exploited by DLS. Thus other naming schemes are possible as well.

As DLS is part of CORBA, we have to decide how the newly loaded code should be represented in a CORBA environment. We decided to represent the code as CORBA object. We do not require that this object is activated at some object adaptor. So, it may not be accessible from remote sites. On the other hand, it is possible to activate it, e.g., by passing its reference via a POA using the implicit activation policy.

Usually this code object comes to life as a side effect of a request to DLS. However, DLS may decide to re-use code objects that have already been requested. On the other hand, DLS may create a new object when the implementation of the code was updated, e.g., for bug fixing. Furthermore we anticipate that the code object is rather stateless. In fact, it may have state, but it should not matter whether DLS creates a new code object or re-uses an existing one if they both are representations of the same functionality, and have the same symbolic name respectively. A typical application is to use the code object as a factory for CORBA objects and values, or for other objects of the language environment. Therefore, we refer to the code object in the following as "factory" object.

The service itself is represented by a pseudo CORBA object. This object is always local to the current execution environment similar to the POA. An object reference to the service object can be retrieved by invoking **`resolve_initial_references`** at the ORB object. The reference must be named by the string **`DynamicLoadingService`**. The DLS pseudo object has a very simple interface that allows clients to dynamically request for new code. A single method named **`getFactory()`** receives a symbolic name referring to a certain functionality. The service will now, entirely hidden from the client, retrieve a corresponding code implementation that fits best into the current execution environment. After having loaded the new code it is converted into a CORBA object. A local reference is created and passed back to the caller.

The caller has to know the precise type of the factory object to narrow the reference for further use. Otherwise, the client may want to deploy the dynamic invocation interface (DII) to access and use the factory. We anticipate that for a large group of applications a simple pre-defined factory interface will do. Only in cases where special parameters for object creation are needed object-specific factories are to be defined.

If the requested functionality could not be loaded due to some failure an exception is thrown and no object reference is returned. There can be a variety of reasons why a loading request may fail. One reason may be that the name is not connected to any implementation. In this case an exception named **`InvalidName`** is thrown. Another reason may be that the service is currently not available due to network or service-internal problems. In this case an exception named **`ServiceNotAvailable`** is thrown. All other reasons imply that the service was not able to find a code implementation that fits into

```
module dls {
      enum PlatformNotSupportedReason {
            format, format_version, mach, byteorder,
            address, os, os_version,
            lib, lib_version, other
      };

      exception PlatformNotSupported {
            PlatformNotSupportedReason reason;
            string reason_message;
      };

      exception InvalidName{};
      exception ServiceNotAvailable{};

      interface DynamicLoader{
            Object getFactory( in string name )
                  raises( InvalidName, ServiceNotAvailable,
                              PlatformNotSupported );
      };
};
```

Fig. 3.1 Interface Description of the Dynamic Loading Service

the current execution environment. In this case an exception named **`PlatformNotSupported`** is thrown.

The **`PlatformNotSupported`** exception is augmented by a reason field that gives a hint why the selection of a suitable implementation failed. We do not anticipate that this hint is of any use for the application program. In very most cases, the application will have to resign and live with not having loaded the code if that is possible, otherwise quit. However, we included the hint as it allows the application to print an error message that may be interpreted at least by system programmers to solve the problem. The different failure reasons are:

- There is an implementation but not in the right code format (**`format`**).
- There is an implementation in the right code format but it has the wrong format version (**`format_version`**).
- There is an implementation that could be loaded but it contains code for the wrong platform (**`mach`**).
- There is a loadable implementation for the current platform but it has the wrong byte order (**`byte_order`**)[1].
- There is a loadable and runnable implementation but it contains code with wrong alignment and addressing scheme (**`address`**).
- There is a suitable implementation but it uses system calls of another operating system (**`os`**).
- There is a suitable implementation for the current operating system but the required version of the operating system is not compatible (**`os_version`**).

1. This error can occur especially for interpreted languages that can store intermediate code in different byte orders.

- There is a suitable implementation for the current system but it requires additional libraries that are not available (**`lib`**).
- There is a suitable implementation for the current system that requires additional libraries; those are available but not in the right version (**`lib_version`**).
- There is a suitable implementation but additional tests executed by the code have failed to confirm compatibility of the code (**`other`**).

The order of the reasons described above implies the order of compatibility checks made by a DLS implementation. If an error message occurs apparently none of the available code implementations has passed all compatibility checks. The error message presents the most specific incompatibility that lets the last available code implementation fail to pass the compatibility checks.

Fig. 3.1 presents the IDL definitions of DLS including the three exceptions. A typical scenario of using DLS in a Java-based execution environment is shown in Fig. 3.2. First, the initial reference to the DLS pseudo object has to be retrieved. Second, the reference has to be converted in a reference to the **`DynamicLoader`** interface using a CORBA narrow operation. Third, the **`getFactory()`** method of the service can be called with a symbolic name of the code's functionality. Finally, the returned reference has to be converted to the known interface type of the factory by another narrow operation.

```
org.omg.CORBA.Object o=
      orb.resolve_initial_references("DynamicLoadingService")

dls.DynamicLoader dl = dls.DynamicLoaderHelper.narrow(o);

org.omg.CORBA.Object o = dl.getFactory("org.mycompany.myapp.func4711");

MyFactory factory = MyFactoryHelper.narrow(o);
```

Fig. 3.2 Scenario of Usage from a Java-Based Environment

4 Example Implementation

We built an implementation of a CORBA dynamic loading service that enables an application to request and integrate new functionalities implemented in Java and native code written in C or C++. Our service implementation is built on top of *JacORB* an open source Java-based ORB [2]. However, our DLS implementation is hardly ORB-dependent and thus easily portable to any Java-based ORB.

The DLS pseudo object that can be received from the ORB object is always local to the current execution environment. This is necessary as the local environment somehow has to integrate the code to be loaded. However, not all components of a DLS implementation need to be local. Some components can be shared by many DLS pseudo objects, and execution environments respectively. The design of our DLS implementation is modular and as system and language independent as possible. Since the dynamic loading of code takes place at runtime, speed is a major concern. The current implementation takes this into account and effectively chooses and loads the right instance of implementation.

4.1 Architecture

Our DLS implementation is split into three parts: the *dynamic loader*, the *code registry* and the *code server*. The *dynamic loader* is the local representative of the DLS service. Its interface is exposed by the DLS pseudo object that can be retrieved from the ORB. The task of the dynamic loader is to select a suitable code implementations, load it and return a reference to the factory object to the calling application. The code registry knows of all loadable functionalities and their implementations. It helps in pre-selecting implementations that can run in the requesting environment. The code server stores code implementations and delivers the one that was finally selected by the dynamic loader. Code server and code registry are built in such a way that only the dynamic loader contains language- and platform-specific parts. Thus, code server and registry can be reused by many execution environments. In fact, these components can even reside on different nodes. Fig. 4.1 shows a setting with three different nodes hosting the three different parts of our DLS implementation.

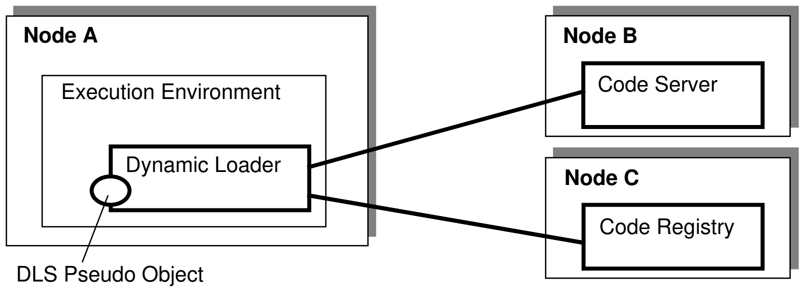

Fig. 4.1 The Architecture of the DLS Implementation

The *dynamic loader* is a core component of the DLS implementation. It is responsible for the coordination of the code-loading process. It receives requests for new functionalities via the **`getFactory()`** method. Each request is forwarded to the code registry augmented with the local system parameters. System parameters include all the static properties about the local execution environment that are potential selectors for implementations, e.g., processor architecture, byte order, operating system, addressing scheme. In our implementation, those values are supplied by the Java runtime environment via system properties. In general, system parameters can be dynamically computed or pre-configured by system administrators.

The *code registry* maintains a database of code descriptions. For each functionality a description entry is stored together with the unique symbolic name mentioned above. A new functionality and the associated implementations (further called *implementation instances*) can be registered with their symbolic name. When a dynamic loader wants to load code the code registry will get the symbolic name and will have to find the corresponding description in the database. If it cannot find a description to a given name an **`InvalidName`** exception will be thrown and eventually forwarded to the client application. Thus, the implementation of a code registry is comparable to that of a CORBA naming service.

A found description entry contains information about all the loadable instances implementing the functionality, including their dependencies, requirements and locations.

Internally the description entry is represented in XML. It is described in more detail in Section 4.2. According to the system parameters passed by the calling dynamic loader, the code registry filters the description entry and returns only information about those implementation instances that are able to run in the calling execution environment. If there is no matching code implementation a **`PlatformNotSupported`** exception is raised and eventually forwarded to the application. This pre-selection reduces the communication overhead between dynamic loader and code registry.

Since the code registry is not directly involved in code loading it does not necessarily need to be local to the requesting execution environment. Instead it could be located anywhere in a distributed system. In our implementation the code registry is implemented as an ordinary CORBA object that is initially known to each dynamic loader[2]. The code registry object is accessed by standard CORBA remote invocations.

From the code registry, the dynamic loader gets a description entry stripped to a subset of the available implementation instances. The deployment of these instances may depend on several conditions that cannot easily be expressed and communicated to the code registry. Those conditions may be

- availability checks of local libraries and
- results of dynamically running and instance-specific tests on the local environment.

Thus, the dynamic loader will check each of the instances for compatibility with the local environment. This process is described in more detail in Section 4.3. The decision to integrate a simple pre-selection into the registry and the more specific selection into the dynamic loader was mainly drawn due to performance considerations. Passing all necessary information to the registry for a pure registry-based selection on one hand and passing the description of all instances to the dynamic loader for a pure load-based selection on the other hand are both wasting bandwidth.

After having selected a suitable implementation instance the corresponding code will be loaded. Usually the code is provided by traditional Web- and FTP-based servers that act as what we call a *code server*. However, there may be other servers accessible by different protocols. As long as they somehow allow to retrieve a code file they can also be used as code servers. There can be a whole set of code servers in a distributed system. The description entry just stores a suitable URL to the file on the code server. After loading the code, the dynamic loader will create the corresponding factory object and return it to the application.

4.2 XML-Based Description of Implementation Instances

As the code registry shall be language and platform independent we chose to represent description entries of functionalities as XML-based text. Also the filtered description entry returned from the code registry to the dynamic loader is represented as XML text. Alternatively we could have defined a complex data structure in CORBA IDL. However, we decided to pass XML text as it is the native form of the stored descriptions. Converting XML into an IDL data structure would add some additional parsing overhead.

2. We anticipate that the code registry is registered with the naming service under a well-known name and can thus be retrieved by dynamic loaders.

The code registry usually has to manage the data of several implementation instances per given functionality. These instances represent independent entities of code which could implement the functionality in different programming languages and for different platforms. Thus, the structure of a description entry represented in XML is preserved by the root tag **`functionality`** which has an attribute **`name`** and could include an arbitrary number of **`instance`** tags. An instance tag includes all necessary information about the instance's requirements and the location of the corresponding code.

The basic requirements are enclosed by the tag **`systemparams`**. Those are the machine architecture, byte order, address length and the implementation languages of the code. We distinguish between a primary and a secondary language environment. The primary language environment is the one that is used for the factory object to be created. The secondary language environment could be used for additional code written in a different language. In our current implementation Java is always the primary language environment. The secondary language environment can be a C or C++ environment running native code. Both environments are specified using the tags **`primary-language`** and **`secondary-language`**. There must be a **`primary-language`** tag per **`instance`** tag. Additionally, there could be several secondary-language tags.

The code of an implementation instance sometimes relies on other code modules which have to be locally available at the hosting execution environment. These extended requirements are described inside of a **`depend`** tag. This tag can enclose tags of type **`resource`** which describe code modules which the instance of implementation relies on. Optionally there can be **`version`** and **`version-range`** tags inside the **`resource`** tag that specify what versions of a code module are required. Finally an instance of implementation could have additional specific requirements. To check if these requirements are fulfilled a compatibility test could be specified with the **`test`** tag. Such a compatibility test is comparable to a very simple instance of implementation and could include almost the same tags as a instance tag.

After the requirements the code location is specified using tag **`code`**. This tag can include an arbitrary number of **`location`** tags that describe the location of a code module and how to access it. Multiple location tags describe alternative locations for fault tolerance and availability. The last tag of an instance description is the **`factory`** tag. It specifies how the factory object can be accessed inside the loaded code. The way how this is done depends on the primary language environment. In our implementation, the factory tag refers to the class name of the factory object that needs to be created.

Figure 4.2 shows an example description of a functionality called **`NativeHello`**. This functionality has only one instance of implementation. The code of this functionality consists of two elements a Java archive and a shared library. The shared library relies on an x86 processor and Linux as operating system. Additionally the instance specifies a test which ensures that a special requirement is fulfilled by the execution environment (e.g., a certain output device is present).

4.3 Code Selection and Loading

If the code registry returns one or more instances of implementation, the dynamic loader has to check local dependencies and extended requirements. First the dynamic loader will check whether all code modules that the preselected instances of implementation

```
<functionality name="NativeHello">
      <instance>
            <sysparams>
                  <primary-language name="java" format="bytecode">
                        <version value="1.3"/>
                  </primary-language>
                  <secondary-language name="C/C++" format="elf">
                  </secondary-language>
                  <mach address="32" byteorder="little" name="x86"/>
                  <os name="linux">
                        <versionrange min="2.2.3" max="2.3.1" />
                  </os>
            </sysparams>
            <depend>
                  <resource format="elf" type="lib" name="libxml.so">
                        <major-version><version value="2.2.10"/>
                        </major-version>
                  </resource>
            </depend>
            <test>
                  <code><location format="..." type="jar" url="..."/>
                  </code>
                  <factory name="hello.Test"/>
             </test>
            <code><location format="bytecode" type="jar" url="..."/>
            </code>
            <code><location format="elf" type="lib" url="..."/>
            </code>
            <factory name="hello.HelloFactoryImpl"/>
      </instance>
</functionality>
```

Fig. 4.2 Example description of a functionality

rely on are locally available and have the required version. Our DLS implementation supports several types of dependent code modules. As the service is implemented in Java dependent modules may contain Java program code. So, dependent code modules can be single class files or entire *Java Archives* (JARs). Whether a certain class is locally available can be easily verified by trying to load the class through the system class loader. If this fails the class is not available. Testing the availability of a Java archive needs a different procedure. The Java execution environment lists all archives and directories which are accessible to the Java language environment in the system property **`java.class.path`**. For checking whether a JAR file is available the dynamic loader has to search for the name in the class-path property. Most JARs contain a *Manifest* file in which all important information about the JAR is stored. This includes the version information of the JAR so that the required version can be checked.

Another type of dependent code supported by our DLS implementation is code implemented as shared library. If a Java program wants to call a native method using the Java Native Interface (JNI) [7] it first has to load the library by using the static method **`System.loadLibrary()`**. The Java runtime environment completes the library name with the operating-system–dependent name extensions and tries to load it via the system loader. The system loader has to know where the libraries could be located. This information is supplied by the system property **`java.library.path`**. To determine if a shared library is locally available the dynamic loader only has to check if a file with the library name is available. This is done by checking all directories included in the **`java.library.path`** for the needed library. To determine whether the needed library is available in the expected version is highly dependent on the operating system. On Unix-based systems the version information is encoded into the file name. So, the dynamic loader has only to check for the according file. All Windows versions support the Dynamic Linking Library format (DLL). This format includes special fields which contain the version information. Thus, the dynamic loader has first to check if the library is locally available and then read the version information of the library.

If the code dependencies are solved the dynamic loader has to check the special requirements of an instance of implementation if there are any. This is done by executing the tests described by the instance of implementation. But before this is possible the code dependencies of the test must be checked and the code of the test has to be loaded from the specified code server. This is a similar process as the loading of an implementation instance. Test implementation instances have a generic test interface including a default method **`testRequirement()`** that is invoked for executing the actual test. If the test method returns true the processing of the instance of implementation continues otherwise a **`PlatformNotSupported`** exception is thrown with reason **`other`**.

If all requirements of an instance of implementation are fulfilled the program code is loaded. In the current implementation the Java code can be loaded by the class **`URLclassLoader`** of the Java Standard Library which offers a simple interface to retrieve Java byte code from HTTP or FTP servers. The retrieval of native libraries is a bit more complicated. The code has to be stored in a local directory which is listed in the system property **`java.library.path`** so that the Java runtime environment can dynamically load these libraries. After all code modules are loaded the factory can be instantiated and returned to the calling application.

Of course every step of the processing of an instance may detect errors. If as an end result no implementation instance passes all checks, a final exception is thrown to the calling application. As the exception signals the reason why the last instance failed to pass the checks the exception is most valuable to the application developer or system administrator.

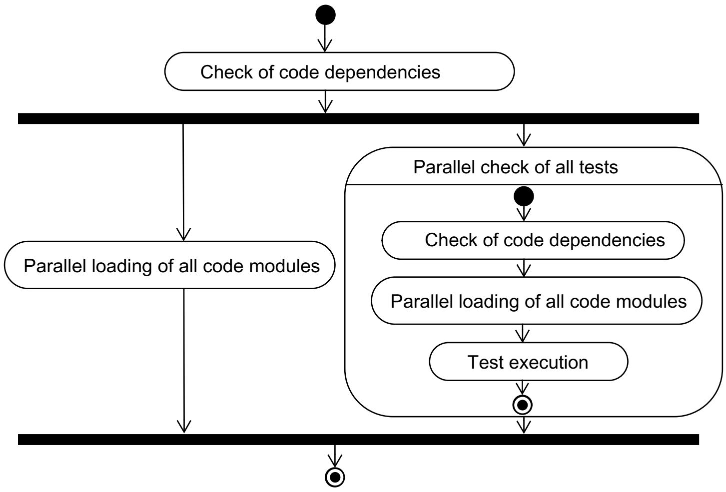

Fig. 4.3 Parallel Processing Model

All the processing could be done in a sequence of single steps. However this is rather time consuming. Since the dynamic loading of code takes place at runtime and speed is a major concern we implemented a parallel processing model. If an application requests a new functionality and the code registry returns one ore more instances of implementation to the dynamic loader the processing of instances executes concurrently. First the code requirements get checked. This could be done quite fast so we need no concurrent execution. The next steps are the checking of special requirements and the loading of code. As the transfer of different code modules from different code servers could be quite time consuming this is done concurrently. If a tests fails or a code server is not accessible all parallel and depending activities are stopped. Already transferred code will be deleted. Fig. 4.3 shows the concurrent execution of code loading and test execution of a single instance.

5 Applications of DLS

This section will briefly outline possible applications of DLS. We will first focus on mobile CORBA objects that can also be used to implement mobile agent. Then, we will look at smart proxy implementations.

5.1 CORBA Lifecycle Service

The OMG defined the so-called lifecycle service for the CORBA architecture. This service specifies how CORBA objects can be created, destroyed, copied and moved [11]. Usually it is up to the vendor of a CORBA implementation how the lifecycle service is supported. We claim that with using DLS the CORBA lifecycle service can more or less have a generic implementation. For brevity we just demonstrate how creation and migration of objects can be implemented with DLS as a basic mechanism.

The lifecycle's abstract model of creation is that there is a factory object which provides a specific method for creating a new CORBA object. The factory object can create the desired type of object on the same node as the factory resides. With using DLS the factory could load the necessary code on demand and then create the object.

The abstract model of migrating an object is more complex. The object itself provides a method named **`move()`** that allows to migrate the object to another node. This node is determined in an abstract fashion. Therefore, a parameter referring to a so-called factory finder is passed to the invocation of **`move()`**. The factory finder specifies the possible locations and returns references to suitable factories that can create the desired object at one of the possible locations. The implementation of **`move()`** now has to select a suitable factory, create a new object, transport the original state to the new object and finally make sure that the new object is accessible with existing object references. The latter is usually done with the help of a implementation repository that authoritatively knows the real communication address of an object. As a last step the old object is deleted.

In both cases, creation and migration, the DLS is deployed inside a factory to load the new code. This may even allow to use generic factories for a set of object types. The lifecycle service specification provides selectors for the desired code, so-called keys [11]. Keys of kind "object implementation" can directly be used as symbolic DLS name to refer to the desired code functionality.

5.2 Intelligent Proxies

In certain cases the client-server style of RPC-based interaction between a local stub and the remote CORBA object is not sufficient for distributed applications. Imagine a CORBA object that provides multimedia data at its interface or a replicated object that has many instances installed in the Internet. A client of these types of objects cannot just use a simple stub but needs a more intelligent piece of local software to receive the multimedia data or invoke methods on many replicas at once. As CORBA does not support this kind of communication some research systems extended CORBA by so-called intelligent or smart proxies. As an example we refer to the Squirrel system [4].

AspectIX is the name of our own CORBA implementation that takes the idea of intelligent proxies to an extreme: all proxies, stubs and servants are considered to be part of a single but fragmented object [3]. As soon as a local *AspectIX* environment gets a previously unknown object reference, a local fragment of that object is installed, usually a stub or smart proxy. As the local environment cannot know the code of all object reference that it will ever use, the code of the local fragment has to be loaded on demand. This is done by using DLS. In *AspectIX* an object reference can include a special *AspectIX* profile[3]. The *AspectIX* profile contains the symbolic name needed by DLS to load the code of the fragment. Additional communication addresses stored in the profile help the newly created local fragment to find the other fragments of the same object and to

3. The Interoperable Object Reference (IOR) of a CORBA object may include multiple so-called profiles for different ways to contact an object.

communicate with them. Inside of the *AspectIX* ORB, e.g., in the **`string_to_object`** operation, the profiles of an incoming IOR are analyzed. In case of a pure CORBA object a traditional CORBA stub is created. If an *AspectIX* profile is found the corresponding code is loaded, a new fragment is created and initialized with the profile's communication addresses.

6 Related Work

We are not aware of any work that has tried to make dynamic code loading available to CORBA applications in a generic way. However, there are of course systems implementing the CORBA lifecycle service, e.g., the *LocALE* system [9]. LocALE provides a framework for supporting the lifecycle of distributed CORBA objects. Objects can even migrate between heterogeneous platforms. LocALE does not address the problem of code migration instead it assumes that the needed code is already installed at the local system. In fact, our DLS implementation could be integrated into LocALE's lifecycle servers to enable dynamic code loading of even system-dependent code.

The *Plug-In Model* [8] is a system that enables the migration of objects through special factories which are not entirely compatible to the CORBA lifecycle service. A so called plug-in server delivers code to a local factory finder. The factory finder creates a factory from the loaded code. The factory is able to create a new object and to manage the state transfer for a migration. The code of the factory has to be provided as a dynamic link library (DLL). The system is thus restricted to Windows-based platforms. However, the factory may be written in a variety of languages. The plug-in server is comparable to our code server since it provides only the code for a requested factory. The factory finder is comparable to a combination of our code registry and dynamic loader. The Plug-In Model does not offer the possibility to choose between multiple implementations and does not allow the selection of system-dependent code.

Software deployment systems can also be considered as related work, e.g., software package management systems. A simple system is the *Red Hat Package Manager* (RPM) [6] used on Linux platforms. It allows to install, update ,and verify software packages. A package provider has to describe the software and its dependencies similarly to our description entries. During the build process of a package the host operating system and the machine architecture is automatically detected and noted in the package. There is no possibility to specify further system requirements like operating system version or the address length. However, the RPM system offers the possibility to specify tests as shell scripts which are executed before the actual install process. Besides, all package managing systems are entirely user driven and only support the static deployment of software.

Another interesting software deployment system is *Java Web Start* which uses the Java Network Launching Protocol [5]. This system describes the code and the requirements of a Java application in a special XML format. So described applications can be installed over the net via the Web Start client. The format is highly Java specific and targeted to install and update software. However it considers the possibility that Java application could make use of native libraries. These libraries are describe in a **`nativelib`** XML tag and enclosed by a resources tag which could have an attributes **`os`** and **`arch`** for the specification of the required operating system and platform architecture. So, system-dependent native libraries can be selected and installed by the Web Start client. The current release lacks the support for dependent resources and for locally executed compatibility tests.

7 Conclusion

We presented the informal specification of DLS, a CORBA service for dynamic loading of code. Our service is platform and language independent. Thus, applications using the service are portable from one CORBA platform to the other as long as the service is available on both platforms. We also outlined a prototype implementation of DLS for a Java-based ORB. This implementation is split into environment- and platform-specific parts (implemented in each execution environment using the service) and sharable parts hosted on one or multiple server nodes. The sharable code registry stores code and variant information, the sharable code server delivers the code. The local and environment-dependent dynamic loader finally selects, downloads and integrates the right code module. Our prototype implementation transparently loads not only Java code but also native code written in C or C++, thus improving even Java's sophisticated standard code loading mechanisms.

Our service is easily configured using XML-based description entries per loadable functionality. Code selection is not only done by matching platform-specific properties (e.g., operating system, processor architecture) but also guided by dynamically loaded and executed test modules that check the compatibility of the local execution environment. This potentially enables all sorts of compatibility tests.

The basic motivation for this research was to support the *AspectIX* ORB that provides a smart proxy-like approach using fragmented objects [1]. However, the DLS service can also be deployed for a generic implementation of the CORBA lifecycle service and for building a mobile agent platform.

Our service implementation needs further improvements. So far we did not integrate any security measures but use the facilities provided by the underlying language environment. However, for a truly generic service this is not enough. Additionally, techniques for scalability and fault tolerance need to be deployed for our code registry (e.g., caching and replication of code modules).

8 References

[1] AspectIX Research Team, Univ. of Ulm, Univ. of Erlangen-Nürnberg. *AspectIX Project Home Page.* **<http://www.aspectix.org>**

[2] G. Brose: "JacORB: Implementation and Design of a Java ORB." *Proc. of the IFIP WG 6.1 Int. Working Conf. on Distrib. Appl. and Interoperable Sys.* – DAIS (Sep. 30–Oct. 2, 1997, Cottbus, Germany). Chapman & Hall 1997.

[3] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, M. Steckermeier: "AspectIX: a Quality-Aware, Object-Based Middleware Architecture." *Proc. of the 3rd IFIP Int. Conf. on Distrib. Appl. and Interoperable Sys.* – DAIS (Krakow, Poland, Sep. 17–19, 2001). Kluver, 2001.

[4] R. Koster, T. Kramp: “Structuring QoS-Supporting Services with Smart Proxies.” Proc. of the IFIP/ACM Middleware Conf. (Pallisades, NY, 3.–7. April 2000). Lecture Notes in Comp. Sci. 1795, Springer, 2000.

[5] R. W. Schmidt: “Java Network Launching Protocol & API Specification (JSR-56) Version 1.0”,Home Page <http://java.sun.com>,Dec. 2000

[6] E. C. Edward: “Maximum RPM: Taking the Red Hat Package Manager to the Limit”, Red Hat Software, Feb. 1997

[7] S. Liang: *Java Native Interface: Programmer's Guide and Specification.* Addison Wesley, June 1999.

[8] C. Linnhoff-Popien, T. Haustein: “Das Plug-In-Modell zur Realisierung mobiler CORBA-Objekte.” *Kommuunikation in Verteilten Systemen* (KiVS), Informatik aktuell, Springer 1999.

[9] D. Lopez de Ipina, S.-L. Lo: “LocALE: a Location-Aware Lifecycle Environment for Ubiquitous Computing.” *Proc. of the 15th Int. Conf. on Information Networking* – ICOIN-15 (Jan. 31–Feb. 2, 2001, Beppu City, Japan).

[10] Object Management Group: *The Common Object Request Broker Architectur and Specification*. Ver. 2.6, OMG Doc. formal/01-12-35, Framingham, MA, Dec. 2001.

[11] Object Management Group: *CORBAservices: Common Object Services Specification.* OMG Doc. formal/97-12-02, Framingham, MA, updated Nov. 1997.