A Generic Infrastructure for Decentralised Dynamic Loading of Platform-Specific Code

Rüdiger Kapitza[1], Holger Schmidt[2], Udo Bartlang[3], and Franz J. Hauck[2]

[1] Dept. of Comp. Sciences, Informatik 4, University of Erlangen-Nürnberg, Germany
rrkapitz@cs.fau.de
[2] Institute of Distributed Systems, Ulm University, Germany
{holger.schmidt,franz.hauck}@uni-ulm.de
[3] Siemens AG, Corporate Technology, Munich, Germany
udo.bartlang.ext@siemens.com

Abstract. Dynamic loading of code is a crucial and often neglected part of today's distributed systems that face increasing dynamics, complexity and heterogeneity. Ubiquitous computing and mobile computing even strengthen this trend. As the local availability of suitable code cannot be assumed in such environments, we propose a generic, decentralised code loading infrastructure. The whole process of publication, look-up, implementation selection and the final loading of platform-specific code is decentralised and requires only basic peer-to-peer functionality. In contrast to previous work, our infrastructure allows any peer participating in the network to offer and to obtain platform-specific code in a dynamic and heterogeneous environment. By building on our generic concept, we present a JXTA-based service for dynamic code loading, which is realised by extending and improving JXTA-built-in mechanisms for dynamic service integration. Subsequently, we show the practical application of our infrastructure by an integration into our CORBA middleware and an implementation of mobile objects and mobile web services.
Key words: CORBA, Dynamic Loading of Code, JXTA, Peer-to-Peer, Web Services

1 Introduction

Distributed applications of any domain face the trend of raising complexity, dynamics and heterogeneity of software and hardware. Two prominent protagonists that emphasise this development are ubiquitous computing [1], targeting distributed applications on small, mostly embedded devices, and planetary-scale execution environments for globally available services such as PlanetLab [2] and Xenoserver [3].

In both cases, applications—especially distributed ones—have the requirement to dynamically load additional code at run-time if that code is not already bound to the local execution environment. There, challenges to dynamic code loading arise if rarely used code has to be loaded on demand or if code to load is not even known in advance. This is a common problem, as distributed applications usually have numerous independently running application parts, which results in some code modules not being known at compile or even at start-up time. However, it is desirable that newly developed

code can be used by already running execution environments. Additionally, for some distributed applications it is not feasible to install and load all code modules at every node of the system. For example, some code modules might only be used by a few of the nodes, and these nodes may not be known in advance or may have resource restrictions.

For addressing these problems, we recently proposed a dynamic loading service that enables the dynamic loading of platform-specific code [4]. However, this work follows a classical client/server-based approach relying on a central component managing metadata of all know implementations and their variants. In contrast to that approach, this paper proposes a generic and decentralised peer-to-peer-based lookup, selection and loading process. This allows multiple parties to independently and non-reliably provide implementations for a certain object or component. Building on this generic concept, a prototype was implemented that uses existing concepts of the JXTA platform [5] to dynamically select and load code based on metadata descriptions called *advertisements*. These advertisements are extended to provide a truly platform-independent support for the dynamic loading of platform-specific code. JXTA is used because of its flexibility: it allows replacing routing mechanisms (e.g., unstructured replaced by structured topology) without having to change the application, i.e. our prototype, itself.

We evaluated the proposed and implemented system by its integration as a common CORBA service to support mobile objects, and ported this approach to a web-service infrastructure. Then, we integrated the infrastructure into our CORBA-compliant middleware Aspectix [6], to extend the support for fragmented objects. Summarising the results of the use cases, the proposed infrastructure meets all demands to dynamically select and load code for CORBA objects, web services, and fragmented objects within heterogeneous execution environments.

In the following section, a platform-independent and decentralised approach to dynamically discovering, selecting and loading platform-specific code is described. Starting from this point, a brief overview of the JXTA peer-to-peer middleware and its facilities for service lookup and integration is given in Section 3. Then, we describe our prototype implementation of a platform-independent peer-to-peer-based loading service. Section 5 outlines two possible use cases of our infrastructure, the integration into the CORBA-compliant middleware Aspectix and the support for the dynamic creation and migration of mobile objects and services. Finally, Section 6 presents related approaches and Section 7 concludes.

2 Generic Decentralised Dynamic Loading of Code

In the following, a generic approach to dynamically loading locally unavailable and platform-specific code is presented. As every functionality might be available in various implementations with different requirements and properties, a generic and decentralised selection process is responsible for identifying the best-fitting one for a certain environment.

2.1 Requirements and Properties for Implementation Selection

We identified three categories of properties and requirements that have to be fulfilled or at least be taken into account during the selection process (cf. Figure 1).

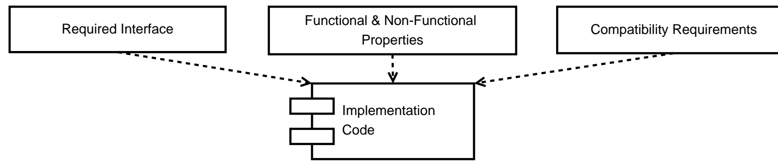

Fig. 1. An approach towards a generic code classification

As an interface determines how the application deals with implemented functionality at the programming layer, new and locally unavailable functionality is identified by its *required interface*. Thereby, the interface has to be defined in a generic interface description language, e.g., using the CORBA Interface Definition Language (IDL) or the Web Service Definition Language (WSDL).

Functional properties express additional functional aspects beyond the bare provision of an interface, e.g., the supported middleware platform. In general, it is hard to standardise all kinds of functional properties. However, this is a requirement for a generic selection process. Thus, we propose that an infrastructure for dynamic loading should specify well-known functional properties and delegate the evaluation of other ones to the application. Implementations providing the same functionality might also possess *non-functional properties* that specify in general quality-of-service properties, e.g., timing behavior and resource consumption of a certain implementation. In the same way as functional properties, these are hard to be standardised in general and therefore might have to be handled by the application.

Specific *compatibility requirements* for a certain implementation have to be considered as well, e.g., the required programming language and execution environment. Such approach considers the fact that exactly the same functionality can be implemented in various programming languages, e.g. in Java or C++, or for specific run-time environments, e.g. Linux or Windows. Compatibility requirements can be automatically evaluated as there is a limited set of properties (e.g., compiler, processor, operating system), outlined in detail in our former work [4], that determine whether an implementation is executable in the context of a requesting application.

2.2 Basic Infrastructure

For dynamic decentralised loading of code, we propose an infrastructure that is composed of three basic components. A *dynamic loader* provides an interface to the application for requesting locally unavailable functionality. This dynamic loader component is able to discover, to select and to integrate an appropriate implementation into the address space of the requesting application. Thereby, the searching process is supported by a *decentralised implementation repository* that stores information about available code implementations. We favour a repository on the basis of a peer-to-peer overlay network, which only has to provide support for keyword search (e.g., JXTA, Gnutella [7]). The implementation repository itself is updated by multiple *code providers*, i.e., peers, that provide implementation code and publish metadata descriptions specifying requirements and properties.

2.3 Basic Data Structure of the Implementation Repository

Using the set of properties and requirements outlined in Section 2.1 enables the selection of the best-fitting implementation code. Therefore, all data about available implementations is published as metadata descriptions in scope of the implementation repository. For omitting duplicated information and improving extensibility, these descriptions are split up into four kinds of metadata, which are each published separately.

An *interface description* contains the fully-qualified name of the interface and the interface (e.g., IDL or WSDL). Within the description, other interfaces and complex data types are also referenced by their fully-qualified names, which enables a dynamic lookup of unknown interfaces and data types.

For covering all interfaces and complex data types of a module, these are combined and published in a *module description*. There, interfaces are only referenced by name. The combination of module and interface descriptions allows a complete representation of the interface description and can be used for providing a decentralised interface repository.

An *extended functional description* specifies all functional and non-functional implementation-independent properties. These are properties provided by various implementations and therefore are used for selecting equal implementations providing the same interface. As mentioned earlier, it is hard to identify a generic set of functional and non-functional properties that apply to a major number of applications. Therefore, an implementation repository and associated dynamic loaders should provide a flexible interface that enables applications to introduce code for custom evaluation.

An *implementation description* describes a concrete implementation and its compatibility requirements. It includes a reference to the location of the code and a description of the initially accessed implementation element. In context of Java this would in general be a class name of a factory.

2.4 Basic Workflow of Publication, Selection and Loading of Code

Before publishing an implementation, a code provider has to generate appropriate metadata documents, i.e., the interface description, the extended functional description (referencing the interface description) and the implementation description (referencing the extended functional description and the concrete implementation). Then, these metadata documents are published via the decentralised code repository.

When an application requires locally unavailable functionality, it passes the fully-qualified name of the required interface and an optional handler for custom evaluation of extended functional requirements to a dynamic loader entity. This dynamic loader requests the implementation repository to look up the interface description and, if not available, passes an exception to the calling application. Then, the repository is queried for extended functional descriptions supporting the requested interface. If provided, the results are passed to the optional handler, which has to return an ordered list of appropriate extended functional descriptions starting with the best-fitting one. On the basis of this list, the dynamic loader queries the repository for implementation descriptions. These are evaluated depending on a policy, e.g., the first

fulfilled implementation description is selected or all are considered and the best-fitting one is selected. After having selected an appropriate implementation description, the code has to be loaded.

3 JXTA and Dynamic Loading of Code

In this section we give a brief introduction to the JXTA platform and present JXTA's facility for dynamic loading of code.

3.1 JXTA Overview

The JXTA project was initiated by Sun Microsystems as an effort to provide a generic and open infrastructure for peer-to-peer computing. For establishing a generic basis for peer-to-peer applications, JXTA standardises fundamental functions by introducing six asynchronous query/response protocols [8].

A JXTA peer-to-peer network consists of *peers* (uniquely identifiable nodes), which syndicate to *peer groups* [9]. These peer groups permit the segmentation of the JXTA overlay and provide a set of services which are represented through *advertisements*, i.e., external programming-language-independent XML metadata representations. In general, the availability of any network resource, e.g., peers and services, is represented through advertisements with a unique identifier, which is published within a certain peer group for a special lifetime [8]. Thus, peers try to discover certain resources by searching for the corresponding advertisements.

JXTA introduces the abstraction of *pipes*, i.e., unidirectional, asynchronous, unreliable and virtual communication channels for peers within the same peer group. The endpoints of a pipe are dynamically bound at run-time, even to different peers. JXTA introduces two different kinds of pipes: a *point-to-point pipe* for unicast communication and a *propagate pipe* for multicast communication.

3.2 Dynamic Lookup and Loading of Services

For structuring and dynamically extending JXTA-based applications the infrastructure offers a generic module framework. *Modules* are managed by the framework and represent distributable units of functionality within a specific peer group that can be initialised, started and stopped by a peer. Thus, modules enable loading and integrating new services into the JXTA platform [10].

For efficiently discovering modules, the definition of a module is divided into three types of advertisements. As JXTA claims to be both language- and platform-neutral, a *module implementation advertisement* enables the differentiation of multiple module implementations, e.g., a module could be implemented in Java or C++. This advertisement specifies implementation-specific details, e.g., the actual code location. For handling different versions of a module, *module specification advertisements* are introduced, which are referenced by corresponding module implementation advertisements. Additionally, a *module class advertisement* announces the pure existence of a unique module class. It provides an abstraction for referring to a module that

provides a particular class of functionality (independent from a certain specification or implementation). As multiple module specification advertisements can relate to a certain module class advertisement, references are embedded into the module class advertisement.

Recapitulating the facts, JXTA allows building a decentralised module taxonomy to support the discovery and loading of services. However, class advertisements only announce the availability of a general category of functionality. This gives developers an idea for a certain module specification and supports the selection process at a very high level, but for an automated module selection process at application level, additional conventions have to be established. Therefore, the Java reference implementation of JXTA makes implicit assumptions that a module implementation provides a certain interface for starting and stopping a module, but this is neither specified by the JXTA protocol specification nor declared by advertisements. Additionally, JXTA offers no support for determining and specifying the interface of a module offered to higher layers like an application. This makes it hard to provide multiple implementations supporting the same protocol for the same platform but providing different properties. Furthermore, module implementation advertisements should enable the providing of compatibility information but are not standardised so far. This results in JXTA implementations specifying their own format and parameters, which prevents the use of module implementations in context of different JXTA implementations. Altogether, the JXTA support for dynamic loading and integration of services leads to platform-specific implementations and does not support dynamic loading of arbitrary code.

4 A JXTA-based Infrastructure for Decentralised Dynamic Loading of Code

Although JXTA's approach for dynamic loading of code seems to be generic and flexible, we outlined its weaknesses and shortcomings. Thus, it cannot be used as a generic and platform-independent infrastructure for dynamic code loading. In this section, we extend this infrastructure based on our generic concept for dynamic code loading, which in general only relies on support for keyword search within a peer-to-peer infrastructure (cf. Section 2).

4.1 Extended Advertisements

We extended the advertisements conforming to our specified requirements in Section 2.3 to provide an own code loading infrastructure on top of JXTA. Figure 2 shows required advertisement types and their relations. In our approach a *module class advertisement* represents the implementation interface. We define that the *name* field of the advertisement specifies the fully-qualified name of the described functionality's most-derived interface. The advertisement's *description* field is used for representing the interface description. As the name field of the class advertisement is indexed in the JXTA network, an interface can easily be searched by its name.

The *module specification advertisement* is mapped to an extended functional description, considering non-functional properties as well (e.g., code-versioning).

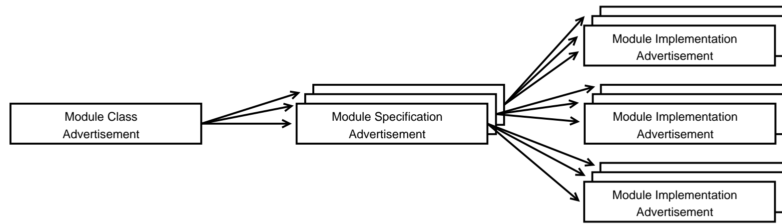

Fig. 2. Relations of extended advertisements

We consider the protocol specification as a functional property that declares if and how a functionality is network-dependent. Additional functional and non-functional requirements are encoded into the *description* field. If the specified functionality is offered by a JXTA service, there is a pipe advertisement for addressing; otherwise the dependent field is left open.

Finally, the *module implementation advertisement* is extended using standardised compatibility requirements that we defined in previous work [4], e.g., system parameters as the used run-time environment. These requirements are stored in the *comp* field. In addition to our former work, we add platform-dependent interfaces to the compatibility requirements. This explicitly allows to specify an integration of certain functionality at platform level. In contrast to the Java JXTA reference implementation that only allows loading a JAR-file from a web server specified within a *puri* field, we provide extended facilities to reference and to transfer a code archive from an arbitrary code provider. Therefore, we embed a module specification advertisement in the *puri* element, enabling the specification of necessary functionality to communicate with a certain code provider. This enables the flexible integration of arbitrary services for the dynamic code transfer as there is no predetermined transfer protocol. A requesting peer is able to dynamically fetch a code transfer service over the peer-to-peer network. For instantiating the service, the main class is specified within the *code* element.

Such code transfer handler should either be offered via the HTTP-based code transfer support provided by JXTA or by the implementation of the basic JXTA transfer service that is described in Section 4.3. Thus, in general we assume at most one level of indirection.

4.2 Decentralised Implementation Repository

Section 3.1 introduced peer groups as a mechanism for grouping users with similar interest. In context of our prototype implementation we use a dedicated peer group (*Code Peer Group*) for publishing and discovering implementations. A *code provider*, which is described in the following subsection, publishes advertisements related to offered implementations within this peer group.

Unfortunately, JXTA binds module specification advertisements to a pipe that is again bound to a certain peer group. The consequence is that this peer group is also used as the group to address the services for execution. If this is not feasible, the dependent

module specification advertisements have to be discovered, modified by providing a group-specific pipe advertisement, and finally republished in scope of the Code Peer Group.

4.3 Code Provider

As described before, JXTA provides only restricted mechanisms for code transfer and sharing. Therefore, we developed an own code provider service, which enables code sharing and transfer via the JXTA network.

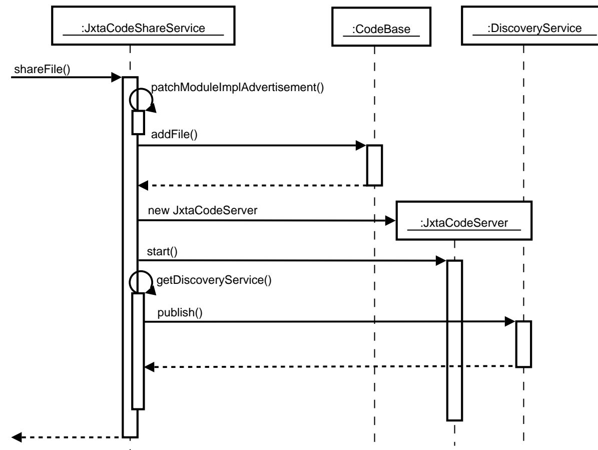

Fig. 3. Code sharing process (UML sequence chart)

Before publishing an implementation and its dependent code archive, associated advertisements have to be generated, if not already available. Therefore, a local `JxtaCodeShareService` object offers the core functionality to publish and to share implementations. Thereby, a code archive together with the three advertisements is passed to the `JxtaCodeShareService` via the `shareFile()` method. Then, the service contacts two other objects as shown in Figure 3. First, the `JxtaCodeShareService` adds its pipe advertisement for code transfer to the module specification advertisement, then it passes the archive to the `CodeBase`. This object administrates the locally offered code archives. Then, an instance of the autonomously working class `JxtaCodeServer` is created, which provides a multi-threaded server that is responsible for the file transfer via a simple JXTA-based protocol. In the last step, advertisements are published via the standard JXTA discovery service.

4.4 Dynamic Loader

The dynamic loader builds the core of our prototype. Figure 4 illustrates the collaboration between its important components. `JxtaCodeHandler` is the central

entity during the whole dynamic loading process. It is responsible for coordination and finally initiates the code transfer.

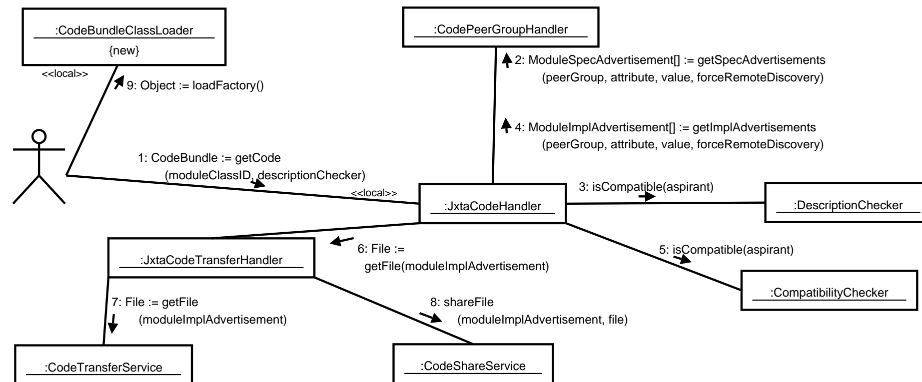

Fig. 4. Collaboration of central system components

The dynamic loader expects only a module class ID to determine the basic interface and additionally the information of a module specification advertisement to determine appropriate functionality. The module class ID can be determined by the application using the fully-qualified name of the most derived interface of the required functionality. The method `getCode()` of the `JxtaCodeHandler` enables searching for a certain module specification advertisement. Therefore, it allows key identifiers as a module class ID, name, version or a generic description within the *desc* element. The latter is achieved by passing an object that implements a `DescriptionChecker` interface that is able to perform a validity test for the concrete use case (1). For selecting a specific implementation code instance, the dynamic loader uses the module class ID for discovering corresponding module specification advertisements within the code peer group (2). Based on the module specification advertisement and the generic `DescriptionChecker`, the discovered specification advertisements can be filtered for a suitable one (3). It might be necessary to start multiple requests to the JXTA network if no suited specification advertisement is available yet. Based on the extracted module specification ID, a search for corresponding implementation advertisements can start (4). The dynamic loader compares received implementation advertisements to requirements of the local execution environment (5): An advertisement is chosen by using an object that implements a `CompatibilityChecker` interface, which is able to validate the suitability for the current execution environment. If a suited module implementation advertisement is found, the `JxtaCodeHandler` is able to initiate the code transfer, if an appropriate transfer handler is locally available (6) (Otherwise, a suited transfer handler has to be fetched recursively). This operation is transparently processed by the `JxtaCodeTransferHandler` (7). Thereby, the `JxtaCodeTransferHandler` encapsulates the whole transfer process by offering a method `getFile()` that only takes a module implementation advertisement as

parameter. If the code transfer to specific provider fails, another code provider could be chosen if available. Exemplarily, a code transfer service supporting file transfer using the JXTA network is realised within our prototype implementation. If the code transfer succeeded, the code can be offered by the requesting peer for supporting the scaling of the whole peer-to-peer system (8). Additionally, persistent caching of the code avoids further remote transfers of identical code resulting from future requests. As a last step, an object-specific factory is used to dynamically integrate the fetched code bundle into the running system (9).

5 Applications of the Generic Decentralised Dynamic Loading Infrastructure

In this section, we present two exemplary applications using our proposed loading infrastructure: integration into a CORBA middleware to support mobile objects/services and support for fragmented objects.

5.1 Supporting Dynamic Loading of Code for Mobile Objects and Services

Recently, we proposed a platform-independent object migration service based on the CORBA Life-Cycle Service (LCS) [11]. The LCS specifies several interfaces for supporting object migration. The migration process is shown in Figure 5. A migratable object has to support the `LifeCycleObject` interface that includes a `move()` method for initiating the migration. Within this method, a target location has to be determined. Such location is represented by a *generic factory*, which enables the creation of objects on remote machines. The selection process of an appropriate generic factory is supported by a *factory finder*.

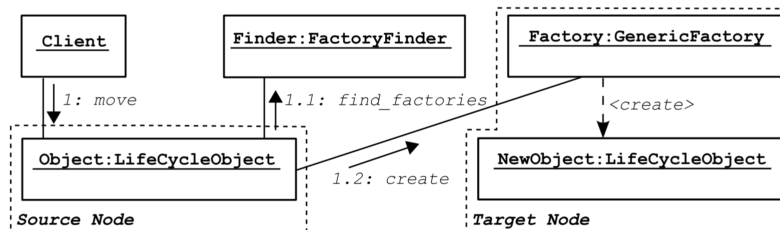

Fig. 5. Object migration based on the CORBA Life Cycle Service

For migrating an object from a source node to a target location, current state and code have to be transferred, as local existence of arbitrary code cannot be assumed. For transferring the state of an object, we use CORBA value types. For code provision, we use our decentralised dynamic loading infrastructure as a CORBA service. Thus, it can be accessed by the generic factory or any other local CORBA application. During

the initialisation phase of the ORB, the local JXTA runtime platform is configured and connected to the peer-to-peer network, which enables the service immediately after the ORB's initialisation. The service interface is equal to the interface of the Dynamic Loading Service (DLS) [4] that offers one central method `getFactory()` for requesting new functionality that is provided by an object implementing the factory pattern. Additionally, our decentralised dynamic loading service offers the opportunity to pass a custom handler for supporting the selection process. The generic factory's `create()` method for object creation expects a parameter with the required interface's fully-qualified name and optionally a selection handler. Thus, the generic factory is able to request platform- and object-specific factories using the `getFactory()` method offered by our service. Figure 6 outlines the core sequence of the loading process performed by the service. First, the `JxtaDynamicLoader` class is invoked for requesting a new implementation and passing a custom object for selecting an appropriate implementation. On success, a `CodeBundle` reference is passed to our custom class loader instance, which offers a method for creating and initialising a requested object implementation, i.e., in case of the generic factory a specific factory that is able to instantiate the demanded object.

```
public Object getFactory (String moduleClassID,
    DescriptionChecker desc){
  try {
    CodeBundle codeBundle = JxtaDynamicLoader.getCodeHandler().
        getCode(moduleClassID, desc);
    CodeBundleClassLoader loader = new CodeBundleClassLoader(
        codeBundle);
  } catch (NoCodeAvailableException e1) { ... }
    catch (MalformedURLException e2) { ... }
  return loader.loadFactory();
}
```

Fig. 6. Dynamic loading of a previously unknown object within the `getFactory()` method

Additionally, we implemented a prototype for migrating a web service. Therefor, we transferred the LCS concept to web services, which results in the factory finder and the generic factory being implemented as web services. The generic factory web service offers a `create()` method, to which the required web service interface is passed (as WSDL). Based on this WSDL description, the generic factory is able to determine the required interface and implementation. In this scenario, the factory directly interacts with our decentralised dynamic loader infrastructure. By using the `getCode()` method, a service-specific factory can be loaded and created. This service-specific factory is able to deploy a platform-specific instance of the required web service at the target location with setting the correct state (transferred from the original web service).

5.2 Enabling Dynamic Binding of Fragmented Objects

The Aspectix middleware provides a CORBA-compliant but more flexible and extensible Object Request Broker (ORB) implementation compared to standard CORBA by building on a modularisation of the handling of object references (IORs). A *generic reference manager* uses *portable profile managers*, which encapsulate all tasks related to reference handling, i.e., reference creation, reference marshalling and unmarshalling, external representation of references as strings, and type casting of representatives of remote objects [6]. Currently, Aspectix provides profile managers for standard CORBA and additionally offers support for the fragmented object model and other non-CORBA middleware platforms, such as Jini or Java RMI.

On the one hand, a fragmented object offers a standard object interface to the outside, on the other hand, a fragmented object can be composed of several fragments and could be distributed with arbitrary internal architecture. This offers a high degree of freedom and flexibity. For interaction with a fragmented object, a corresponding local fragment has to be created that either acts as a simple stub for the fragmented object or as a more intelligent stub that includes parts of the fragmented object's functionality (implicit binding). Furthermore, such infrastructure enables the exchange of a fragment implementation at run-time and leaves the implementation of the internal communication and structure open to the developer. Binding to a fragmented object in general requires dynamic loading of fragment-specific code as it is not predictable if and when a certain fragment implementation is needed. Therefore dynamic loading of code is an essential service to support fragmented objects at their full flexibility.

We extended the fragmented-object-supporting profile manager by using the dynamic loading service outlined in the previous section. The profile of a fragmented object references the initial fragment implementation, either directly by specifying a class name or indirectly by providing a code reference to the standard DLS. Depending on a tag, either the implementation is directly loaded or one of the two code loading services is used (standard or decentralised). In case of the decentralised loading service, the profile includes a module class ID. This enables loading the code of a certain fragment implementation using the JXTA-based dynamic loading infrastructure as described in Section 4.4. After having loaded the code, the fragment implementation has to be instantiated and initialised. As this is a fragment-specific task, every fragment implementation has a standardised constructor that is executed by the profile manager.

The fragmented object model allows an easy integration of arbitrary internal communication patterns. Thus, by building on our dynamic loading infrastructure, we also created a prototype for dynamic selection, loading and integration of peer-to-peer services into a standard-CORBA-compliant middleware [12]. Therefore, based on the support for fragmented objects, we provide a special JXTA IOR profile that contains a module specification advertisement, which contains the service description and the supported protocol. This enables loading the fragment implementations, which are actually represented by JXTA service instances, using our presented decentralised loading service. Such fragmented objects provide a standard CORBA interface to the outside while internally interacting in a peer-to-peer fashion. Through this, the gap between standard client/server-based middleware and the JXTA peer-to-peer infrastructure can be closed.

6 Related Work

In previous work [4], we presented the Dynamic Loading Service (DLS), a CORBA service for dynamic code loading. Similarly to the realised loading service of this work, the DLS permits to load remote code with consideration of the current run-time environment and other requirements. However, the DLS follows the client/server paradigm and uses dedicated servers to host the program code and to offer specific information about available code. In contrast, our current work builds on a JXTA-based peer-to-peer-network.

Another interesting system is Java Web Start [13]. This software deployment system uses the Java Network Launching Protocol and describes the code and the requirements of a Java application in a special XML format. This results in applications that can be installed over the net via a special Java Web Start client (even system-dependent native libraries can be selected and installed). However, the format is highly Java-specific, aims at installing and updating software and the current release lacks the support for dependent resources and for locally executed compatibility tests.

The OSGi service platform [14] defines an open run-time environment, enabling dynamic service integration. For the bundled representation of a service's functionality, the concept of an OSGi bundle is defined. A special characteristic of such a bundle is the possibility to be dynamically added and removed from the run-time environment. Compared to this work, a bundle offers extended possibilities, in order to specify dependencies of other services. However, the OSGi approach misses sophisticated mechanisms for describing, remotely discovering and selecting code portions as outlined in this work. Furthermore, OSGi primarily targets at code loading and sharing for the Java programming language, whereas our approach is generic and can be applied to other programming languages as well.

Paal et al. proposed a distributed code loading infrastructure based on multiple application repositories that can be dynamically queried by a custom application loader [15]. In contrast to our approach, this system offers fine-grained code loading based on *class collections*, which are represented by class subsets of a Java archive. However, the system is limited to the Java programming language and application repositories have to be preconfigured at initial deployment time for enabling code loading.

A peer-to-peer-based architecture for remote loading of Java classes is described in [16]. This approach shows an alternative way to the standard Java class loader mechanism and is exemplarily realised using JXTA. Compared to our solution, it lacks flexibility to describe and to search for suitable program code. Thus, the architecture neither permits a representation of loadable code with the JXTA concepts of module advertisements nor offers support for a custom transfer protocol.

7 Conclusion and Future Work

We presented a generic and decentralised approach to dynamically discover, select, load and integrate platform-specific code. According to the common peer-to-peer idea, every peer within our infrastructure is able to load code and, additionally, to provide this code on demand. Our prototype implementation extends and improves the mechanisms for

dynamic service integration of JXTA. However, the proposed generic concept can be applied to any peer-to-peer infrastructure that at least supports keyword-search. For evaluating the dynamic loading infrastructure, we presented exemplary applications.

Security issues are beyond the scope of this paper. Dynamic loading of code always involves security considerations, and we assume that standard security mechanisms such as code signing and a public-key infrastructure can be used for securing our peer-to-peer-based dynamic loading service. Additionally, JXTA enables restricted groups, in which only authorised peers are able to participate. Thus, a general trust between users can be achieved using such mechanism. However, our implementation does not yet make direct use of such techniques.

Even though our prototype supports the precise selection of platform-specific code, we currently assume that a concrete implementation is more or less self-contained. This means, that either necessary libraries are at the target platform, as described by the compatibility requirements, or included in the dynamically loaded code archive. We therefore would like to provide support for implementations that reference other interfaces or implementations that should be loaded dynamically.

References

1. M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3), 1991.
2. L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. *SIGCOMM Comput. Commun. Rev.*, 33(1):59–64, 2003.
3. E. Kotsovinos, T. Moreton, I. Pratt, R. Ross, K. Fraser, S. Hand, and T. Harris. Global-Scale Service Deployment in the XenoServer Platform. In *1st Works. on Real, Large Distrib. Sys.—WORLDS'04*, San Francisco, CA, December 2004.
4. R. Kapitza and F. J. Hauck. DLS: a CORBA service for dynamic loading of code. In *OTM Confederated Int. Conf.*, Sicily, Italy, 2003.
5. L. Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5(3), 2001.
6. F. J. Hauck, R. Kapitza, H. P. Reiser, and A. I. Schmied. A flexible and extensible object middleware: CORBA and beyond. In *5th Int. Works. on Softw. Eng. and Middlew.* ACM Digital Library, 2005.
7. T. Klingberg and R. Manfredi. Gnutella 0.6. Technical report, 2002.
8. The Internet Society. Jxta v2.0 protocols specification. Technical report, Sun Microsystems, 2001.
9. Sun Microsystems. Jxta v2.3.x: Java programmer's guide. Technical report, 2005.
10. B. J. Wilson. *JXTA*. New Riders, jun 2002.
11. R. Kapitza, H. Schmidt, and F. J. Hauck. Platform-Independent Object Migration in CORBA. In *OTM Confederated Int. Conf.*, LNCS 3760, pages 900–917. Springer Verlag, Oct 2005.
12. R. Kapitza, U. Bartlang, H. Schmidt, and F. J. Hauck. Dynamic integration of peer-to-peer services into a CORBA-compliant middleware. In *OTM 2006 Workshops*. Springer, 2006.
13. Sun Microsystems. Java Web Start Overview. White paper, 2005.
14. The OSGi Alliance. OSGi service platform: Core specification, release 4. Technical report, 2005.
15. S. Paal, R. Kammüller, and B. Freisleben. Dynamic software deployment with distributed application repositories. In *14. Fachtagung Kommunikation in Verteilten Systemen (KiVS)*. Springer, 2005.
16. D. Parker and D. Cleary. A p2p approach to classloading in java. In *2nd Int. Works. on Agents and P2P Comp.—AP2PC'03*, 2003.