Platform-Independent Object Migration in CORBA

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

[1] Dept. of Comp. Sciences, Informatik 4, University of Erlangen-Nürnberg, Germany
rrkapitz@cs.fau.de
[2] Distributed Systems Laboratory, University of Ulm, Germany
{holger.schmidt, franz.hauck}@uni-ulm.de

Abstract. Object mobility is the basis for highly dynamic distributed applications. This paper presents the design and implementation of mobile objects on the basis of the CORBA standard. Our system is compatible to the CORBA Life-Cycle–Service specification and thus provides object migration between different language environments and computer systems. Unlike others, our Life-Cycle–Service implementation does not need vendor-specific extensions and just relies on standard CORBA features like servant managers and value types. Our implementation is portable; objects can migrate even between different ORBs. It supports object developers with a simple programming model that defines the state of an object as value type, provides coordination of concurrent threads in case of migration, and takes care of location-independent object addressing. Additionally we seamlessly integrated our implementation with a dynamic code-loading service.

Keywords: Object Migration, Platform Independency, CORBA, Life-Cycle Service, Value Types, Dynamic Loading of Code.

1 Introduction

One of the key features of object-based distributed programming environments like CORBA (Common Object Request Broker Architecture) is the transparent access to remote objects. The middleware infrastructure hides the distribution and the heterogeneity of the underlying computer hardware, operating system, and programming language. However, full access transparency is not always useful. Sometimes the true distribution of objects should be visible and controllable by applications. Examples are applications that explicitly move distributed objects for balancing load, for handling failures, and for minimizing communication overhead (e.g., mobile agents). For mobile objects the middleware system has to support state transfer and location-independent addressing of objects. Often, the support mechanisms are tightly woven into the middleware and therefore highly system dependent.

CORBA is ammended by the Life-Cycle–Service specification [1], which describes a service concept based on common design patterns to implement object

R. Meersman and Z. Tari (Eds.): CoopIS/DOA/ODBASE 2005, LNCS 3760, pp. 900–917, 2005.

mobility and other life-cycle operations. Objects have to implement a special Life-Cycle interface that, among others, provides a `copy()` and a `move()` method to duplicate and migrate an object. Although the Life-Cycle–Service specification defines the general life-cycle process, it has certain shortcomings that lead to unnecessary burdens for application programmers and to system-dependent and incompatible implementations. We propose the design of a generic Life-Cycle–Service implementation, which is only based on common CORBA features and therefore vendor independent. Our prototype is implemented in Java, but can easily be ported to other CORBA-supported languages. Mobile objects can even migrate between different ORBs when those run an implementation of our service design.

For the application programmer, we provide a value-type–based state-transfer mechanism. This frees developers from writing their own state-exchange mechanisms for every mobile object. Furthermore we provide mechanisms for dynamic loading of code based on previous work [2]. Thus, the code for mobile objects needs not to be statically deployed. Finally our implementation encapsulates the coordination of life-cycle operations and frees the application programmer from location management. Our implementation either forwards requests or uses a lightweight location service.

The next section gives a brief introduction of the Life-Cycle–Service specification, its shortcomings, and existing implementations. In Section 3, we discuss different solutions for collecting and exchanging the state of a CORBA object. Section 4 describes the design of our generic Life-Cycle–Service implementation. The development process of a life-cycle object is illustrated by a simple example application in Section 5. Section 6 is devoted to performance evaluations. Finally, Section 7 discusses related work and Section 8 gives our conclusions.

2 CORBA Life-Cycle Service

CORBA is a standardized architecture defined by the Object Management Group (OMG) that allows programmers to create and access objects deployed in a distributed system. CORBA also specifies a set of CORBA services. These services represent optional ORB extensions and address general needs of CORBA applications. The Life-Cycle Service [1] is such a CORBA service, as it enables application-controlled mobility and other life-cycle operations. In the following sub-sections we will describe the core components of the Life-Cycle Service, discuss its weaknesses and shortcomings, and close with a brief overview of existing implementations.

2.1 Basic Functionality

Standard CORBA provides distribution transparency. With the help of an implementation repository migration of objects can also be made transparent [3]. A servant has to be registered at the implementation repository, which from this time on takes care that the servant remains accessible. This mechanism

allows restarting of server processes at different locations, but is not suited for application-controlled object mobility. Additionally, these implementation repositories are tightly woven into an ORB and its dependend POA[1] implementation [4].

In contrast, the Life-Cycle Service allows an application to control the distribution of objects, e.g., creating, removing, copying, and moving of objects. This is especially useful for mobile applications (e.g., mobile agents) and for the management of applications that needs to distribute objects across different platforms for non-functional reasons like scalability and fault-tolerance.

It is assumed that object creation is performed using factory objects. These can also be remote allowing for remote object creation. A factory is a CORBA object offering a method for creating new instances of a particular object type at a particular location. It is not specified how factories are requested to create a new object. This is left to the object developer as there can be different parameters required for different object types. However, the Life-Cycle–Service specification defines an IDL interface named `GenericFactory`. This interface contains a generic `create_object()` operation, which gets a set of criteria represented as sequence of name-value pairs in the IDL type `Criteria`. The Life-Cycle–Service specification gives hints on how to use criteria but does not define any standards. For a specific factory implementation they can be used to select the required object type, object capabilities, different object initializations, and even different locations. The latter can be accomplished by forwarding the creation request to a more specific factory object at a particular location depending on a particular criterion.

```
interface LifeCycleObject {
    LifeCycleObject copy( in FactoryFinder there,
                          in Criteria the_criteria )
        raises( ... );
    void move( in FactoryFinder there,
               in Criteria the_criteria )
        raises( ... );
    void remove()
        raises( ... );
};
```

Fig. 1. IDL specification of the `LifeCycleObject` interface

While object creation is handled by a factory, all other life-cycle operations are executed at the object itself. Therefore, an object supporting the Life-Cycle Service has to implement the `LifeCycleObject` interface (see Fig. 1). The `copy()` operation creates a copy of the object at some location. As a result, a reference to the newly created object is returned. The `move()` operation moves the object to another location; the `remove()` operation deletes the object.

[1] POA = Portable Object Adapter.

Both `copy()` and `move()` need some notion of location in order to place a copy or the object itself. The Life-Cycle Service specifies a `FactoryFinder` interface for objects representing an abstract location. Taking migration of an object as an example, Figure 2 shows the first phase of the interaction between object, factory finder and factory. For duplication of objects the scenario is almost identical.

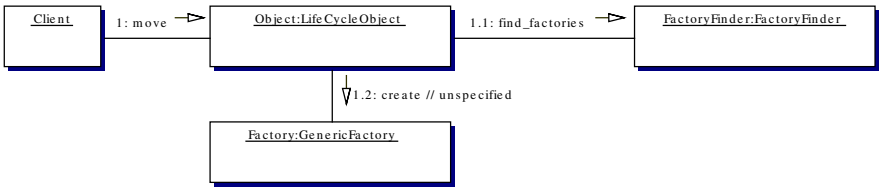

Fig. 2. First phase of object migration (UML collaboration diagram)

First a `move()` method is called on the object implementing the life-cycle interface. An instance of `FactoryFinder` is passed as parameter. The `move()` operation is supposed to ask the factory finder for a factory that finally can be used to create another instance of the original object at a certain location. In form of a key parameter, `move()` can ask for specific factory properties. The key is some sort of name-value pair that was originally introduced for naming objects [5]. Once again, the specification gives hints on how to use the key parameter but does not standardize anything. Anyway, from the key parameter the finder has to select a suitable factory.

The factory is a sub-type of interface `Factory` that remains unspecified[2]. Thus, the `move()` implementation has to know the expected type and has to cast the factory reference to that type by a narrow operation. The factory can have type `GenericFactory` and the criteria set passed to `move()` can be used to influence the factory in creating the object. In the end, `move()` can create a new instance, transfer the state of the original object to the new one, and finally take care that the original object reference remains valid, now referring to the newly created object.

2.2 Open Issues and Shortcomings

The Life-Cycle Service is just a specification. Although the interfaces are specified in detail, the flow of control is just roughly described and implementation details are left to the object developer or the service provider. On one side, this allows for individual implementations of the specified interfaces, as the OMG deliberately underspecified certain issues in order to get them solved by an actual implementation. On the other side, it is likely that Life-Cycle–Service implementations become system dependent and incompatible.

[2] In fact `Factory` is just an IDL typedef to `CORBA::Object`, which has subtle differences to a sub-type.

There are a number of problems with the specification. First, there is no concept specified how the `FactoryFinder` locates existing factories. It is, however, possible to use a Naming Service to retrieve an object by using the key parameter as a name. In summary, the configuration of factories and factory finders is outside of the specification and has to be done by developers. Second, after migration of an object all references to this object should stay valid to maintain location transparency. Unfortunately, the precise procedure for solving this problem is left to the service implementers. Third, the service specification assumes that at each location the required code of the object servant is available. In dynamic environments with mobile objects, it would be preferable to be able to transfer and load code on demand. In a scenario with mobile agents, we cannot assume that the agent code is present at every possible location. Fourth, the most severe problem with the specification is that it does not provide any measures for state transfer. It is neither specified how to determine and gather the state nor how to do the transfer. Usually it is left to the object developer to write the corresponding code. Finally, the specification does not deal with any kind of coordination of concurrent threads. Multiple threads may invoke life-cycle and normal operations that in turn may interfere with state collection and transfer.

In total, we believe that there are too many unnecessary burdens left for the application developers. Additionally, the individual solutions of service developers to the above-mentioned problems make it hard to port life-cycle objects from one system to the other. Migration between different ORBs is usually impossible.

2.3 Implementations of the Life-Cycle Service

There are numerous commercial and non-commercial ORB implementations around that offer facilities and interfaces for application developers to copy or migrate CORBA objects. But either these solutions are platform-dependent like those provided by omniORB [6], ACE ORB (TAO) [7], or the Plug-In Model [8], or they even do not support the Life-Cycle–Service specification at all and provide a totally vendor-specific solution like VisiBroker [9]. None of these implementations addresses platform-independent migration of state and code in heterogeneous environments.

In [10], Peter and Guyennet propose a generic solution for object mobility in CORBA based on the Life-Cycle–Service specification. They focus on coordination of object access during and immediately before and after migrations to increase availably. The object state has to be described as an IDL structure that is used to generate special state-carrying objects with custom access methods. Forwarding is realized using tool-generated proxy objects on the client and the server side that use the CORBA Naming Service as location service. This imposes special actions on the client side. Furthermore, the implementation does not address the provision of platform-dependent code.

Choy et al. describe a CORBA environment supporting mobile agents based on mobile CORBA objects [11]. Based on the Life-Cycle Service and the Externalization Service a concept was mooted, but apparently not implemented.

3 State Transfer in Heterogeneous Environments

As described in Section 2.2, the migration of an object requires the transfer of its state. In homogeneous and more or less platform-independent environments like Java the execution environment may already provide serialization mechanisms. State transfer in CORBA is more challenging, since the state of an object may be transferred between different language environments, i.e. from C++ to Java or Cobol. In this case, language-dependent solutions will fail.

There should be a language-independent and fairly abstract transfer format. For example, it does not make sense to convert the content of a Java `Hashtable` object into a transfer format, as there are about 30 internal and implementation-dependent variables that can hardly be restored in a C++ implementation of that hash table. Instead, it is necessary to distinguish between the state of a particular object implementation and a more or less implementation-independent state that is essential for the object semantics. For a hash table the abstract state will only contain the stored key-value pairs. This abstract state can hardly be automatically identified; instead this has to be done by the developer. Finally, it is useful to define the abstract state in a format that can easily be converted into all supported programming languages so that object developers immediately can identify the transferable object state.

As already mentioned, the Life-Cycle Specification does not specify how to collect and transfer state. Instead, it suggests letting the developer use either a proprietary solution or the CORBA Externalization Service [12]. A proprietary solution may be appropriate if mobile objects move within a homogeneous environment as language-based serialization mechanism can be used. In case of heterogeneous language environments, the object developer needs to find an individual solution for transfer of state, which is likely to be complex and error-prone.

The CORBA Externalization Service was developed to support writing an object state into a data stream and reading that state back from a stream. Whereas this was basically designed for persistence the same concept can be used to support state transfer by shipping the externalized stream to another location and internalize that stream back into another object. Although the Externalization Service offers a common data format this approach has some serious drawbacks. The developer has to write his own marshalling and unmarshalling procedures that call the right operations of the Externalization service in the specific order. This has to be done for every language that is used. In principle, it would be possible to describe the abstract state in some language-independent format and automatically generate the marshalling procedures. However, to date there are no known tools for generating those procedures.

A promising and obvious approach is the description of the transfer state via IDL. Peter and Guyennet [10] used an IDL struct type and provided tool-generated wrapper objects with access methods. This is quite complex and the developer has to implement the invocations of those methods. Instead, we want to support a more generic approach of state transfer that unburdens the developer from calling serialization and deserialization methods at all. For state transfer, we propose IDL value types, a well-known part of the CORBA specification [13].

A value type is similar to an IDL struct, but it can also have methods much like CORBA objects. CORBA objects are declared with IDL interface types. Passing a CORBA object to a possibly remote method transfers the reference to this object (call-by-object-reference semantics). In contrast, passing value-type objects leads to a complete copy of the value type at the receiving side (call-by-value semantics). Like CORBA objects, value types also support inheritance.

Transfer of a value type is realized by transparent marshalling and unmarshalling of the state of the value-type object. In a heterogeneous system it is possible to rebuild value types implemented in one language in another one, e.g., from Java in C++.

```
interface Account{ ... };
valuetype AccountContainer supports Account {
    private float account_state;
};
```

Fig. 3. IDL value type declaration with the supported interface

Like CORBA objects, value types are able to support a specific IDL interface (see Fig. 3). This implies that methods specified in the supported interface are implemented in the value type. A value type supporting an interface can be activated at a POA, and is then remotely accessible. With activation a value type behaves as an ordinary CORBA object that can be passed by reference. Nevertheless it is also possible to pass the value type by value, creating a copy of the value-type object.

For state transfer, we are using value types that support a particular IDL interface. Object functionality has to be encapsulated in a value-type implementation. Thus, the public and private members of the value type represent the abstract state of the object, and the supported IDL interface represents the object's remote methods (cf. Fig. 3). Activated at a POA, the value type works as an ordinary CORBA object. In case of a state transfer, we just pass the underlying value type with call-by-value semantics to another location, which will marshal and unmarshal the necessary state. The object implementation is usually determined by factories registered at the ORB.

To sum up, value types are perfect candidates for implementing state transfer. Value types are well known to CORBA developers since they are part of IDL. Value types can implement CORBA objects and be values at the same time. They document the state and allow the IDL compiler to automatically generate all necessary serialization and deserialization procedures; developers do not have to program them any longer. In the next section we will show how value types can be used in conjunction with a specialized factory to design a generic Life-Cycle Service.

4 Design of a Generic Life-Cycle–Service

This section proposes our generic Life-Cycle Service based on value types and describes the specific implementation details.

4.1 Finding and Selecting an Appropriate Factory

The first step before actually copying or moving an object is the selection of an appropriate factory. The application controls the selection process by passing a factory finder and so-called *criteria* parameters to the life-cycle operation. For the management of multiple factories, we supply a basic factory finder, which will match the needs of most applications. It extends the specified interface by methods for registering and removing factories and other factory finders. The latter enables the common CORBA approach of federations to gain scalability and flexibility.

If a life-cycle object calls the `find_factories()` method, our factory finder looks for matches in the local factory registry. The specification proposes two possible types of factories: specific factories and generic factories. Specific factories support only one type of object; generic factories can support multiple types of objects. To determine if a generic factory is able to create a certain object type it provides a `supports()` method. The factory finder will collect matching factories from its local registry and from all registered factory finders. In turn, these finders can also manage other finders and so on, building a hierarchal federation of finders. After the finder has passed the matching factories, the life-cycle object has to select the appropriate factory based on the criteria parameters and additional object-specific requirements. As this is object-specific it is supposed to be implemented by the developer.

The selection process of the appropriate factory, however, is supported by the generic factory interface as explained in Section 2. A generic factory provides a `create()` method that takes two parameters: a key referencing the object type and a criteria parameter. If the object type is not known to the object or the criteria cannot be satisfied, an exception is thrown.

We provide two variants of generic factories: a single-type and a multi-type factory. The single-type factory can only instantiate a single object type but offers the possibility to check criteria special to this factory and object type. The multi-type generic factory represents a registry for single-type generic factories. As an additional benefit the multi-type factory can process general criteria before even asking other factories. This way, general criteria can be validated that apply to all factories managed by a multi-type generic factory (e.g., checking resource requirements).

For both types of generic factories we provide a basic implementation that only requires an object which implements our `CriteriaChecker` interface. This interface offers a single method `checkCriteria()`, which is executed at the beginning of each creation request. It throws an appropriate exception if the criteria requirements either could not be met or are simply invalid. If no exception is thrown, the creation process proceeds. The single-type factory represents simply a facade implementation of the basic factory, which is explained in detail in Section 4.3, with an extended interface. So criteria could be handed over to the creation methods.

4.2 Coordination of Life-Cycle Operations

During a life-cycle operation, the access to an object has to be coordinated and restricted. For consistency, all three life-cycle operations need exclusive access to

the object in the sense that all earlier invocations have terminated and all others are blocked until the life-cycle operation is executed. Implementing custom coordination mechanisms at the object level could ensure this, but would require deep understanding of application and life-cycle functionality. Therefore, we decided for a solution transparent to object developers.

The first implementation alternative is a specialized POA that controls the access to servants representing life-cycle objects, but a modified POA would be an unwanted ORB-specific extension. Another alternative is the usage of a POA manager, because it can control the state of a POA and block incoming calls via the `hold_requests()` method. Unfortunately, it turns out that this does not work as `hold_requests()` cannot be safely called from a life-cycle operation[3]. Alternativley, portable interceptors represent a central point in the ORB architecture where all incoming and outgoing calls can be caught and modified. A life-cycle object could be registered at a special interceptor at creation time. The interceptor can analyze invocations and take care of proper coordination. However, intercepting all incoming calls is very expensive, as it slows down every method invocation even of non life-cycle objects.

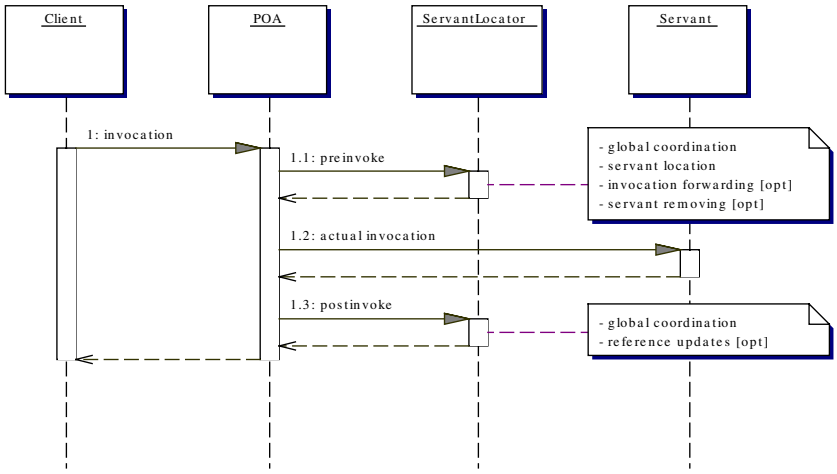

Fig. 4. Operating sequence of an incoming invocation (UML sequence diagram)

We decided in favor of a fourth alternative, a servant-locator–based approach. A servant locator is a special form of a servant manager that is responsible for the activation and management of servants. On every incoming call the POA notifies the servant locator by calling its `preinvoke()` method. The servant locator now has to locate or set up an appropriate servant and return it to the calling POA. After the method invocation on the servant, the POA calls the servant locator again by invoking the `postinvoke()` method. This triggers the locator to tear down the servant or to do other management tasks (Fig. 4).

[3] The calling operation would try to wait for its own completion.

For the access coordination, we implemented a special servant locator that encapsulates the access management on behalf of the life-cycle objects. Every life-cycle object is registered on creation at the locator[4] and owns a synchronized invocation counter, which is managed by the locator. If a method of a registered servant is called, the `preinvoke()` method is executed and the counter is incremented. After the actual invocation, the `postinvoke()` method decrements the counter of the servant. This way all pending calls are accounted.

A life-cycle operation can be detected by the servant locator because the method name of the servant invocation is passed to the `preinvoke()` method. If there are currently other pending calls, the life-cycle call is suspended until all others invocations were finished. Then, the life-cycle operation can be executed. Other incoming invocations—either normal or life-cycle operations—are suspended until the current life-cycle operation will have finished.

4.3 Creation of a Life-Cycle Object

The creation of a life-cycle object being supported by a special kind of factory is a key point in our design. As mentioned earlier, factories support the creation of objects at remote sites on behalf of a life-cycle operation. The factory design pattern is also very useful at creation time of an object. It encapsulates and hides setting up the environment for a life-cycle object and therefore reduces the programming effort for an application developer. Since these tasks are very similar for all life-cycle objects we implemented a general *base factory*. This factory only needs to know the name of the implementation classes to set up a new life-cycle object. The actual creation process has four steps:

1. Instantiation of a servant
2. Activation of the servant
3. Creation of a CORBA reference
4. Registration at a location service (optional)

The first step is straightforward. If one of several `create()` methods of the `BaseFactory` is called, the appropriate instances for the value type are created[5]. An explicit creation of a value-type object is only necessary if the object is completely new (e.g., by calling `create()`). In case of a migrated or copied object a value-type object that encapsulates the object state is passed to a `createFromValueType()` or `createCopyFromValueType()` method. Afterwards the value-type object is activated as a servant at the servant locator. In the next step a CORBA reference has to be generated. On initial creation of a life-cycle object a unique random object id is assigned. This id remains stable for the whole lifetime of the object even in context of `move()` operations. To ensure this and to free the application developer from unnecessary programming effort, every value-type object has to inherit from `MobileContainer`. This value type provides two

[4] Our implementation also allows the registration of application-defined servant locators that are used as delegates from our locator.

[5] In Java a separate Tie class is needed additional to the value-type implementation.

attributes, one for the object ID and another for the forwarding mechanisms. On creation of a new object, a new ID (UUID) is generated and set. In context of a `move()` operation the id can be read from the transferred value type. After the reference is generated, the optional registration at a location service takes place. This is covered in more detail in Section 4.5.

4.4 Dynamic Code Provision

Up to now, our implementation of a platform-independent Life-Cycle Service does not address the mobility of code. In dynamic environments, however, it is often required to transfer not only the state of an object but also the code implementing that object. The CORBA specification provides a code-base parameter for value types to dynamically load code on demand. Unfortunately, the specification suggests that the code base references directly the code of one or more implementations. This is sufficient if an object moves between homogeneous environments, but restricts flexibility if the value type is moved between different language environments.

We provide a special generic multi-type factory that is based on our *Dynamic Loading Service (DLS)* [2]. This service offers the dynamic loading of platform-dependent code on demand for arbitrary functionalities in an ORB-independent fashion. If the generic factory is asked whether a certain type is supported or if a creation of a previously unknown type is requested, the DLS will be queried. If an appropriate object implementation for the current platform exists, the DLS will dynamically load the code and instantiate the associated factory.

For using the DLS, there needs to be a DLS implementation that can be plugged into the local ORB. As DLS is portable to different ORBs this is not a problem. If a DLS is not available, object developers have to link the necessary code into the local system, as in most other Life-Cycle–Service implementations.

4.5 Forwarding and Locating

The reference to a life-cycle object should be valid for its entire lifetime even after migration. Our implementation addresses this problem in two ways: We provide a forwarder-based approach, where the servant locators at previous locations will cooperate in locating objects, and a location-service-based approach, where such a service simply keeps track of the actual object location.

The forwarder-based approach is the default, as it requires no additional preparations and services. The migration of a life-cycle object is initiated by calling the `move()` method. Inside the `move()` method the appropriate factory is selected and invoked. The factory returns a reference pointing to the new location of the object. This new location has to be announced to our servant locator. This is done indirectly by setting the `location` field of the base value-type `MobileContainer`, which every life-cycle object has to extend. After the `move()` operation, this value can be read by the locator inside the `postinvoke()` method. If the move operation fails due to unavailability of an appropriate factory, the location field will not be modified. The servant locator can detect this and the local object remains active.

The location value is registered in a special forwarding table. As already described in Section 4.2, after a life-cycle operation all waiting calls are resumed. Instead of actually invoking the methods on the local object which has moved, the calls are forwarded to the new location. This can be easily done, as the `preinvoke()` method offers a way to throw a forwarding exception with the new object reference. The client-side ORB handles this exception transparently by reissuing the request to the new location. Further requests are automatically forwarded to the new location as the ORB remembers location changes. This approach is very simple and has almost no additional overhead (just maintaining the forwarding table). It requires, however, that the object adapters at previous locations stay up until the object is finally removed. Another downside is a potentially long forwarding chain. Binding to a very early address could cause the first request to be forwarded many times tracking down the route of the object to the current location. More severe is that this method fails if only one of the hosts in the chain crashes or is down for some reason.

To avoid those drawbacks we implemented a simple location service as an additional solution. The key idea is to replace the references provided by our factory implementations. Instead of returning the actual object reference, it is modified to refer to a location service first. This way the location service receives the first invocation after an object is bound and forwards it to the actual location. In order to seamlessly integrate such a service into our implementation, it provides a CORBA management interface for registering and updating locations of life-cycle objects. On creation of an object the factory has to register the object at the location service and modify the returned reference to refer to the location service.

The service itself is also implemented as a servant manager, usually in a separate server process. Instead of locating or setting up the requested servant, the servant manager simply throws a forward exception referring to the actual location of the object. As previously noted this exception is transparently handled by the client-side ORB and the request is invoked on the actual location of the object. If the object moves to another location the factory registers the new location of the object at the service. After a move operation, the old servant locator throws a forwarding exception referencing again the location service which forwards the request to the actual location of the object. If the object is moved and the previous server is no longer accessible, the client-side ORB will fall back to the initial object reference also referring to the location service, which by then knows the new location of the object. To avoid a single point of failure for all life-cycle objects and for scalability reasons, our implementation is able to use multiple location services.

5 Example Application

In this section we demonstrate the implementation of our Life-Cycle Service by a simple application. For the development of a life-cycle object, the following steps have to be performed:

1. Development of the IDL description of the object interface
2. Development of the state description by defining an IDL value type that implements the object interface
3. Implementation of the value type
4. Instantiation of the object with our `BaseFactory`

Our example is an `Account` object described in IDL, which implements simple bank-account functionality. As shown in Fig. 5, this `Account` interface has to inherit from `LifeCycleObject`. The specified methods are object-dependent and implement the needed account functionality.

```
interface Account : ::CosLifeCycle::LifeCycleObject {
    float getBalance();
    void deposit( in float value );
    void withdraw( in float value );
};

valuetype AccountContainer :
        ::org::aspectix::services::lcs::MobileContainer
        supports Account {
    private float balance; ...
};
```

Fig. 5. `Account` interface and the corresponding value type (IDL)

In a next step the IDL description of the value type actually implementing the appropriate object functionality has to be specified. In this value-type declaration also the state has to be specified using private or public data members. The value type has to inherit from `MobileContainer` as described in Section 4.3. Furthermore, it also has to support the previously specified IDL interface. In the example, we declared a value type supporting the `Account` interface (Fig. 5).

The private variable `balance` implements the actual state of the `Account` object, namely the balance information. This state will be transparently transferred via the call-by-value semantics of the value type in case of a move or copy operation (cf. Section 4).

From IDL an abstract `AccountContainer` class is automatically generated. We have to implement a concrete class `AccountContainerImpl` containing all methods of the interface and value type. Of course, the `LifeCycleObject` methods have to be implemented, too. As our Life-Cycle Service takes care of coordination and request forwarding, the actual implementation is rather simple. As we cannot show examples due to length restrictions, we roughly sketch their implementation: In the `move()` and `copy()` method, the developer just has to call `find_factories()` at the `FactoryFinder` and select the intended factory. Finally, the creation method on the factory has to be called. Thus the developer can influence the process of selecting an appropriate factory. The code of the `remove()` method just has to decide whether the object can be deleted or not. If no exception is raised by the operation,

the object will be automatically removed by our servant locator after the execution of this method returns. Within this method the developer is able to do application-specific tasks like deleting external files, etc.

All objects have to be created with our `BaseFactory`. This ensures the necessary POA policies, transparently involves a location service, and reduces development efforts. The value-type object might be created directly in the `BaseFactory` or it might be created and passed to the factory's `create()` method. To run our example application, a `FactoryFinder`, the factories and if necessary a location service have to be started on different machines.

6 Measurements

As our implementation delays calls due to coordination efforts even in cases of no migration, we performed different measurements for estimating the performance penalty of our approach. In another series of measurements we compare different methods of state transfer. All measurements were performed on Intel Xeon 2.4 GHz machines having 2 GB of RAM. We used Java JDK 1.4 and JacORB version 2.2. Effects caused by just-in-time compilation and other run-time optimizations in the JVM have been smoothed out.

6.1 Overhead of Migratable Objects

We first compare the implementation of a CORBA object using a value type with the standard implementation of a CORBA object based on the generated skeleton code. The measurements were performed on two different ORB implementations, JacORB [4] and Sun's built-in ORB from the JDK.

Table 1 shows the results: The first line shows the time needed for a local call sent to a standard CORBA object. The next line presents the time needed for an object implemented by the value-type approach. In the following lines the difference to the standard case is shown, e.g., on JacORB the overhead per call is about 130 ns or 2.3%. In the last three lines we added our servant locator, which has to detect life-cycle operations, coordinate invocations and maintain forwarding if necessary.

Table 1. Difference of time needed for a call using JacORB and SUN ORB

Variant	JacORB	SUN ORB
Standard Skeleton	5.67 μs	4.74 μs
Activated Value-Type	5.80 μs	4.78 μs
Overhead compared to Standard	0.13 μs 2.3 %	0.04 μs 0.8 %
Life-Cycle Object (Value Type and Locator)	9.65 μs	11.88 μs
Overhead compared to Standard	3.98 μs 70 %	7.14 μs 151 %

Our measurements did invoke ordinary operations but still the detection of life-cycle calls and the check for a necessary forward to another location takes considerable time. The implementation uses a Java `Hashtable` that may be the bottleneck. However, we have not yet optimized the implementation for performance.

The measurements based on the Sun ORB show similar results. Compared to the JacORB the value type is much faster, and the locator takes considerably more time. As our locator is exactly the same the additional time is consumed inside of the Sun ORB, but we did not yet investigate where.

6.2 State Transfer

In the next step we compared different methods of state transfer: Java serialization, IDL struct and value type. In all three scenarios we use JacORB and a CORBA remote invocation between a client and a CORBA object. Client and server systems are connected via switched 100 MBit Ethernet LAN. The transferred state contains two long values, two strings containing in total 13 characters and a small octet sequence about 26 bytes.

Table 2. Difference of time needed for state transfer (using JacORB)

Variant	JacORB
Java Serialization	880 μs
IDL Struct	960 μs
Overhead compared to Serialization	80 μs 9.1 %
IDL Value Type	1,150 μs
Overhead compared to Serialization	270 μs 31 %
Overhead compared to IDL Struct	190 μs 20 %

The first test uses Java serialization on the client side to convert the object state into a byte array that is passed by value to the server where it is deserialized. Note that we used a CORBA method call here to pass the serialized data. The measurement serves for comparison with the CORBA-based state transfer techniques and basically shows how much overhead can be saved in a homogeneous environment. The second and third measurements show the invocation time when transferring state by using an IDL struct and a value type. Both objects are reconstructed at the server side without further computation. In both cases, the code is already deployed. Table 2 shows the complete measurements including overheads compared to serialization.

The overhead of a value type compared to a struct is relatively large because JacORB uses reflection to instantiate the particular implementation class of the value type whereas in case of a struct the implementation class is hard-coded into the demarshalling operation. Still there are possible optimizations that we have

not yet investigated in detail. The value-type approach has the additional advantage that the state is already in place at the member components of the value type. With the struct approach the application usually has to access the data in the struct. Using Java serialization is only slightly more efficient and cannot cope with heterogeneous platforms. Of course the precise performance figures depend on the size of the state being transferred. We expect that the larger the state the less dominant the overhead of a value type compared to a struct will be. On the other hand, Java serialization will always be more efficient, but cannot deal with heterogeneous environments.

7 Related Work

As already stated in Section 2.3, there is no implementation of the CORBA Life-Cycle Service that addresses the platform-independent object migration of state and code in heterogeneous environments as our proposed solution does. Only Peter and Guyennet offer in [10] a generic solution for object mobility based on the Life-Cycle Service. However, their solution requires special client-side proxy objects that forward requests. This way, a client has to be aware of life-cycle objects. The object state has to be described as an IDL structure that is used to generate special state-carrying objects with custom access methods. Our value-type–based solution also requires the declaration of the state via IDL, but is more convenient for developers since they need not to call the access functions of custom objects on serialization and deserialization of an object. Furthermore, the implementation of Peter and Guyennet does not address the provision of platform-dependent code.

In the past, a lot of mobile agent systems have been developed, like MOA [14], Mole [15] or Aglets [16], to name a few. But they all provide only mechanisms for migration in homogeneous environments. An Agent Transport Service (ATS) was specified in [17]. There, the Life-Cycle Service was considered, but finally sorted out in order to support lightweight agents. All migration methods are offered entirely by the platform; the agent developers do not have to write any code for this purpose. The Object Management Group (OMG) developed a standard for agent communication: the Mobile Agent System Interoperability Facility (MASIF) [18]. The CORBA Life-Cycle–Service was considered, but as many agent platforms are not based on CORBA, they created their own methods for migration. As an example, many systems use Java Serialization, which can be deployed for homogenous environments only.

In [19] a platform-neutral approach of agent migration is presented. Instead of transferring the code just a blueprint is transmitted. This is possible by assuming that an agent consists of different components. For creating such an agent an *AgentFactory* is specified that creates an executable agent consisting of the right components from such a blueprint. Migration is thus based on transferring the blueprint and the state of an object. The same approach could be layered on top of our Life-Cycle–Service implementation by structuring a complex application as a set of objects each representing a component that are moved together as it is proposed by the CORBA Relationship Service [20].

Finally the usage of value-type objects in our Life-Cycle–Service implementation bares some similarities with the functionality of the Freeze Evictor of the ICE middleware [21]. There the ICE equivalent of a CORBA value type is used to store objects in persistent storage and to access them as remote objects at the same time.

8 Conclusions

We presented a platform-independent implementation of the CORBA Life-Cycle–Service. Although our prototype is implemented in Java it can be easily ported to all CORBA-supported languages and then be used in different ORBs, as it is based on standard CORBA and does not need any ORB-specific extensions.

For state transfer even in heterogeneous environments, we introduced value types. The state can easily described in IDL, and developers do not need to implement state transfer routines. Furthermore, we also provided a solution to code provision in heterogeneous environments. Based on previous work, we offer an optional multi-type–supporting generic factory that loads platform-specific code on demand. This allows the dynamic instantiation of previously unknown objects. Finally our implementation presents flexible mechanisms to provide persistent object references in context of object mobility.

Apart from the fact that the current implementation has already reached a mature state, there are still possible extensions. We currently do not address how to secure the life cycle operations. This can be achieved by doing authentication either at the transport level with the Secure Sockets Layer protocol or at the object level. We also plan a service implementation in C++ and Python, but do not expect any implementation problems. Finally, on top of the current service implementation a mobile agent facility could be established. This would allow for agents that switch the implementation language while moving.

References

1. Object Management Group (OMG). Life Cycle Service Specification. OMG Document formal/2002-09-01, 2002.
2. R. Kapitza and F.J. Hauck. DLS: a CORBA service for dynamic loading of code. In *Proceedings of the OTM Confederated International Conferences*, Sicily, Italy, 2003.
3. M. Henning. Binding, Migration, and Scalability in CORBA. *Communications of the ACM special issue on CORBA*, 41:67–71, 1998.
4. A. Bendt et al. JacORB 2.2 Programming Guide, 2004.
5. Object Management Group (OMG). Naming Service Specification. OMG Document formal/2004-10-03, 2004.
6. S.-L. Lo D. Grisby and D. Riddoch. The omniORB version 4.0 User's Guide, 2004.
7. D.C. Schmidt. Real-time CORBA with TAO (The ACE ORB), May 2004.
8. C. Linnhoff-Popien and T. Haustein. Das Plug-In-Modell zur Realisierung mobiler COBRA-Objekte. In *Kommunikation in Verteilten Systemen*, pages 196–209, 1999.
9. Borland Inprise. VisiBroker for C++ 4.5 - Programmers Guide. Technical report, 2001.

10. Y. Peter and H. Guyennet. Object mobility in large scale systems. *Cluster Computing*, 3(2):177–185, 2000.
11. Breust Choy and Magedanz. A CORBA Environment Supporting Mobile Objects. Technical Report White Paper Draft Version 1, IKV++ GmbH, 1999.
12. Object Management Group (OMG). Externalization Service Specification. OMG Document formal/00-06-16, 2000.
13. Object Management Group (OMG). Common Object Request Broker Architecture: Core Specification, 2004.
14. W. LaForge D.S. Milojicic and D. Chauhan. Mobile Objects and Agents (MOA). In *4th USENIX Conference on Object-Oriented Technologies and Systems*, pages 179–194, Santa Fe, New Mexico, 1998.
15. J. Baumann M. Strasser and F. Hohl. Mole: A Java based mobile agent system. *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, 1997.
16. D.B. Lange and M. Oshima. Programming and Deploying Java Mobile Agents Aglets, 1998.
17. C.A. Mendez and M. Mendes. Agent migration issues in CORBA platforms. In *The Fourth International Symposium on Autonomous Decentralized Systems*, pages 332–335, Tokyo Japan, 1999. IEEE.
18. D.S. Milojicic et al. MASIF: The OMG Mobile Agent System Interoperability Facility. In *Mobile Agents: Second International Workshop, MA'98*, volume 1477/1998, page 50, Stuttgart, Germany, 1998. Springer LNCS. 1477.
19. F.M.T. Brazier et al. Agent factory: generative migration of mobile agents in heterogeneous environments. In *Proceedings of the 2002 ACM symposium on Applied computing* , pages 101–106, Madrid, Spain, 2002. ACM Press.
20. Object Management Group (OMG). Relationship Service Specification. OMG Document formal/2000-06-24, 2000.
21. The Internet Communications Engine (ICE), 2005.