A Framework for Adaptive Mobile Objects in Heterogeneous Environments

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

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

Abstract. The majority of object migration systems do not support heterogeneous environments. Few systems solve this challenge by specifying a platform and language independent state transfer format, requiring a compatible implementation for every target language. However, fields of research like Ubiquitous and Pervasive Computing with mobile users and applications demand an even more platform-independent, flexible and adaptive approach.
This paper presents a novel approach for adaptive object and agent migration in heterogeneous environments based on our former work enabling language- and platform-independent object mobility in CORBA. By providing flexible mechanisms to reduce, expand and transform an object's state and functionality during migration, we support adaptation to the context and application-specific demands at the target system.
This is achieved by introducing a separation of state, functionality and implementation code instead of mapping particular state on particular code. Our prototype system supports object migration from Java to C++ and vice versa. In principle, our concept can be transferred to any CORBA-supported programming language.

Key words: Object Migration, Object Adaptation, Platform Independency, CORBA, Life Cycle Service, Value Types, Dynamic Loading of Code

1 Introduction

Nowadays, computers surround us almost everywhere. This trend is even pushed by the idea of Ubiquitous Computing where computing is integrated into the environment instead of using particular devices. For fulfilling this task, these devices are linked with one another, which should enable an autonomous reaction to particular situations. Accompanied by this evolution and the rising diversity of systems, new concepts and techniques for providing adaptive and context-aware applications are required. Often, these applications should migrate between different platforms during their lifetime. As a typical example, a "follow me" application [1] (e.g. personal information manager application) can have a different interface and state on a desktop computer, a PDA (might also be a smartphone) or a public web terminal. In the first case, the full set of features is provided, whereas in case of the PDA some data-intensive parts are left out. The PDA might even abandon communication-intensive parts, as communication is expensive and restricted in some areas, respectively. Finally, in the web terminal's case, one would only provide the essential parts of the state to fulfil the demanded task for privacy reasons. In all cases, different hardware and software systems can be expected. In other words, we assume that a mobile application has to adapt its state, the provided functionality and the implementation basis to its execution context, the target system and application-dependent restrictions.

Furthermore, there is a need for frameworks supporting complex mobile processes consisting of long-lasting activities. These activities are spread over the network altogether fulfilling the mobile process's task. E.g., Kunze et al. [2] propose a process description language and an execution model for such tasks. However they do not address context-aware and platform-dependent adaptation, which is often required in heterogeneous environments.

Recent object-based middleware and agent platforms restrict migration support to a particular programming language and environment. Only a few systems provide support for migration in heterogeneous environments such as AgentFactory [3] and ATS [4]. These rely on a custom language-independent serialisation format for state transfer and require an implementation providing the same functionality in every target language. A custom state format results in high implementation costs in terms of porting the system to a new platform and providing a fully-fledged implementation of a mobile object for each supported target language. This is especially the case if only parts of the functionality should be provided or are required on certain systems. In scenarios similar to the "follow me" example, which require the ability to dynamically adapt the provided functionality and state according to the current context, a more platform-independent, flexible and adaptive approach for object migration is required.

In this paper, we propose the concept of adaptive mobile objects. These are capable of adapting their state, functionality and underlying code basis during migration to the requirements of the target platform and the needs of the object itself. We focus on weak migration, which means that only the state of an object is migrated but no execution-dependent state, e.g., values on the stack and CPU registers. Our solution is based on CORBA value types [5], a standard CORBA mechanism for passing objects by value, and a dynamic adaptation service for mobile objects, the adaptive object migration service (AOM). Additionally, we provide support for mobile objects acting as mobile agents (an object having an own thread that executes autonomously on behalf of a user). By building on our recent platform- and ORB-independent implementation of the CORBA Life Cycle Service [6] we provide support for heterogeneous environments. In fact, our current prototype supports the migration of objects between Java and C++. Supporting other CORBA languages requires only moderate implementation effort.

The paper is structured as follows: In Section 2, we give an overview of our recent realisation of the CORBA Life Cycle Service. Based on this, in Section 3, we present our concept of adaptive object migration for mobile objects. Then, the development process of an adaptive mobile object is shown by an example application in Section 5. In Section 6, we show related work, and finally, we conclude and discuss future work in Section 7.

2 CORBA Life Cycle Service

CORBA is a standardised architecture defined by the Object Management Group (OMG). This architecture enables programmers to create and access objects deployed in a distributed system, and provides platform and language transparency. Common middleware tasks like object location, request marshalling and message transmission are performed by the Object Request Broker (ORB). A server object is specified by describing the object's interface using the Interface Definition Language (IDL). This interface is used to generate platform-specific stubs for the client and skeletons for the server side. Both entities act as surrogates dealing with heterogeneity and handle remote invocations and marshalling for the object. Objects are actually implemented by servants that are registered at an object adapter. For invoking remote methods, a client only needs a valid object reference that can be bound by the local ORB instance.

CORBA specifies a set of CORBA services. These services represent optional ORB extensions and address general needs of CORBA applications. The Life Cycle Service [7] is such a CORBA service, as it enables application-controlled object management including mobility and other life-cycle operations.

The CORBA Life Cycle Service describes the interfaces of all required components for enabling object mobility in detail. However, the specification leaves open important issues. For migrating a mobile object, the state and the code have to be transferred (this implies determining the state of the mobile object dynamically at runtime). In principle, the specification describes these processes as an object-specific task that should be handled by the object developer. This has various disadvantages as state transfer methods have to be implemented from scratch for each object class. This is error-prone and leads to serious interoperability problems in heterogeneous environments.

However, we identified the fact, that migration in heterogeneous environments enforces differentiating between implementation-dependent and -independent state. A `java.util.Hashtable` object from the Java class library contains many implementation-dependent state variables, e.g., a set of diverse constants regarding the used hash-function. In heterogeneous systems, it does not make sense to transfer state that is highly specific for a certain implementation, as other implementations are not able to interpret these values. This is especially the case if an object is migrated between different programming languages (enabled by our implementation of the CORBA Life Cycle Service). In such systems, just the implementation-independent state should be transferred; in case of a `Hashtable`-Object, this state is represented by the actual <*key,value*>-pairs. To sum it up we consider

- *Implementation-dependent state* as values that are specific for a certain implementation variant of an interface, and
- *Implementation-independent state* as values that are needed by any possible implementation to provide the functionality defined by an interface and values that would be considered as information loss if they were omitted.

Recently, we proposed a platform-independent implementation of the CORBA Life Cycle Service specification [6]. Although our initial prototype implementation is based on Java, it is easily portable to any other CORBA-supported language and can be used in different ORBs as it just relies on standard CORBA without any ORB-specific extensions. This has been verified by a second and interoperable implementation in C++.

For state transfer in heterogeneous environments we introduced value types. Value types are a well-known part of the CORBA specification [5]. In contrast to standard CORBA objects, value type objects are copied by value (call-by-value parameter semantics), leading to a complete copy of the value type at the receiving side. As value types actually represent abstract data types [8], they enable an easy description of an object's *implementation-independent* state using standard IDL syntax. Additionally, developers do not have to write special methods for state transfer, as value type objects are marshalled and demarshalled automatically by the ORB. Like standard CORBA objects, value type objects can also be activated. In this case, a value type has to support a standard CORBA interface, which enables a call-by-reference parameter semantics.

Thus, weak heterogeneous object migration can be realised using the CORBA Life Cycle Service in combination with value types. Fig. 1 shows the basic workflow of an object migration according to the Life Cycle Service specification. The specification defines a `LifeCycleObject`-interface that is implemented by our mobile object. The mobile object is actually a value type that supports the `LifeCycleObject`-interface. Beside other methods, it offers a `move()`-method, which implements object migration in conjunction with a remote factory (`GenericFactory`), that enables the creation of objects on remote servers. For this process, another entity—a `FactoryFinder`—is needed for searching for an appropriate factory on a remote server. Possible locations are represented by a particular `FactoryFinder`. Appropriate locations are selected by evaluating parameterised criteria that restrict possible target locations. For fulfilling this task, the `FactoryFinder` has to maintain a repository of possible factories within the `FactoryFinder`'s domain. Therefore, our factory enables registering and deregistering factories including descriptive metadata. After having found such a factory, the object aka value type can be passed by value as a method-parameter using standard marshalling and demarshalling mechanisms. The created value type has to be activated by the factory using the original object's object-identifier. Finally, the original object is removed.

As we also intend to support heterogeneous systems, in which the object's code might not be present at the final location, we provided a solution to dynamic code provision as well. Based on our previous work, the Dynamic Loading Service (DLS) [9], we developed a special type of `GenericFactory` that transparently

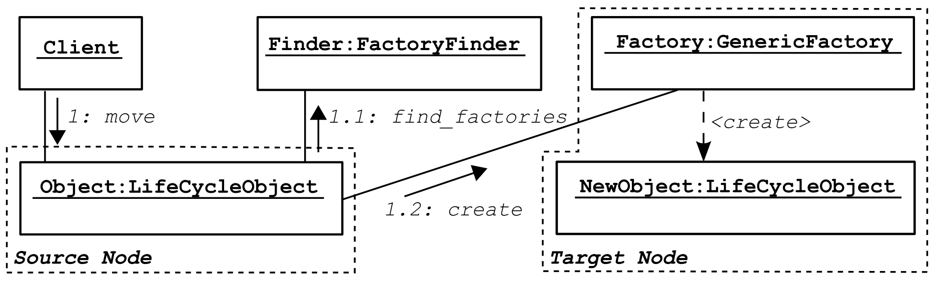

Fig. 1. Object migration using the CORBA Life Cycle Service

enables loading of appropriate platform-specific code on demand. This allows dynamic instantiation of previously unknown objects.

Additionally, our solution enables persistent object references by maintaining the object identifier for the whole lifetime of a mobile object. We achieve this by providing a special type of location service that manages the system's object references.

3 Adaptive Object Migration

In this section, we present our approach of object migration supporting adaptation of state, functionality and code. We give basic background information and outline our basic concept.

3.1 Basic Definitions

For transferring a mobile object in homogeneous environments, usually the state and the code of this object have to be transferred. Therefore, recent systems only separate the state and the code of an object. For addressing heterogeneous platforms, we introduce another abstraction: the separation of state, functionality, and the realising code. Thereby, the functionality of an object is defined by its most derived supported interface. This enables a selection of an appropriate implementation based on a supported interface as already described in Section 2. We call these triples of state, functionality and code the *facets* of the mobile object during its life cycle.
We identified the fact that in various scenarios only parts of the object state are accessed, can be available or should be accessible. If a mobile object moves from one host to another one for fulfilling various steps in a complex workflow, there is data, which is not needed on every target platform. Thus, there is state that we call passive on these platforms. Going back to the "follow me" example from the introduction, the whole state of the personal information manager (PIM) application can be considered as active state on a laptop, as all data can be accessed and modified. As this application object moves to a PDA, certain data

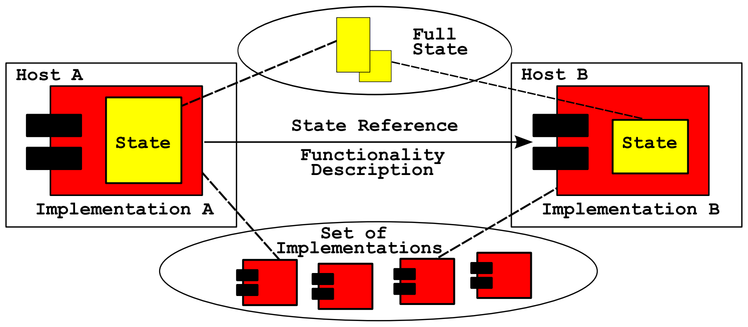

Fig. 2. Adaptation of state according to an object's functionality

items might not be accessible due to device limitations, like private movies or pictures from the last holiday. In this case, the movies and pictures are passive state that is not available. If the PIM application is accessed from a public web terminal, huge parts of state might not be transferred due to privacy reasons. Thus, this information can also be considered as passive state as it should not be accessible. We define active and passive state as follows:

- *Active state* is represented by the state variables that are used and needed for fulfilling the object's functionality at a certain location.
- *Passive state* is the state of an object that is not needed, not available or not accessible at a certain location.

We believe that in all these three exemplary cases of the PIM application not only the state of an object changes or should be changed but also the interface and the implementation that are provided at the target platform. Fig. 2 shows the adaptation of state and interface on demand according to an object's functionalities' requirements. For this purpose, parts of the full state might become passive. However, this passive state might become active again, whenever the object migrates to another location. Thus, even the passive state has to be stored, as it might be used again later.

3.2 Basic Concept

Fig. 3 shows our concept of adaptive object migration (AOM). We enable adaptation of an object's state according to the required functionality as specified by the interface (cf. Section 3.1). In contrast to previous work, we enable attaching state to a special functionality instead of attaching an object's state to a special implementation (represented by code). For migrating a mobile object, just a reference to the state and a description of the required functionality have to be transferred to the new location (cf. Fig. 2). At the new location, our *transformer factory* is able to create an object that fulfils these requirements containing the active state, which is specified by the required functionality. Our infrastructure

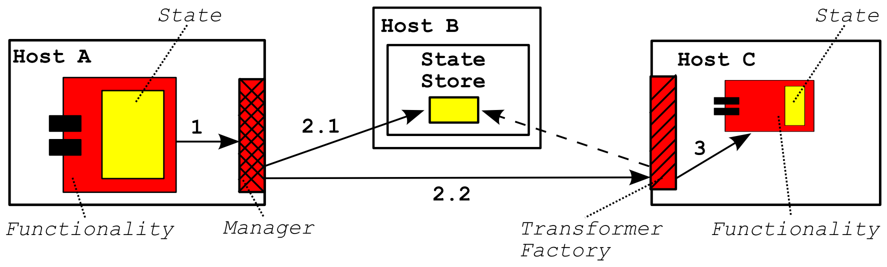

Fig. 3. Basic concept of the adaptive object migration mechanism

supports passivating parts of the state in a *state store* that might be local or remote regarding the current object. Therefore, a *manager* saves the object's current state to the state store (cf. Fig. 3, 2.1). Then, information referencing the state and required functionality is transferred to the target platform (2.2). There, a transformer factory creates the object that is adapted according to the functional and state needs (3). The target side allows receiving and storing the complete state for further local adaptation steps. This requires a local state store and makes especially sense on devices with reduced communication resources and devices that are temporarily disconnected (e.g., a laptop).

As we focus on a mobile environment with resource-limited devices, we support mobile agents. These agents behave autonomously, i.e., they migrate on their own, which reduces resource costs caused by managing the mobile object during its lifetime. Thus, beside a pure migration support, these agents need a facility to get restarted on the target side. We support this by providing a mechanism that automatically invokes a user-defined asynchronous method for agent initialisation upon migration completion. Furthermore, beside supporting migration that is triggered externally, we enable internally-triggered migration allowing agents to autonomously migrate to another node during their life time.

4 Implementation

In this section, we present our implementation for realising our introduced concept for adaptive object migration.

In a first step, the developer has to decide which kinds of functionalities should be supported. As mentioned before, these represent the different facets, which the mobile object can adopt during its life cycle (this affects the behaviour of the object but not the actual object identifier; cf. Section 1).

For any functionality, an IDL interface and the corresponding value type (supporting the interface and specifying the active and implementation-independent state) have to be specified. For supporting our adaptive object-migration service the value type has to support the `LifeCycleObject` interface as well. Additionally, the value type has to inherit from an `AOMObject`[3] value type. This ensures that the value type contains administrative state needed by our infrastructure. Among other things the object's universally unique identifier (UUID), a reference to the local manager object and a reference to the state store.

Transition of state between the diverse facets is realised by a simple name matching algorithm, i.e., if a state variable has the same name in different value types with the same object identifier, then we assume that these state variables represent identical data. Thus, defining value types containing states with identical names but different types results in an error. A value type's state might become active and passive during runtime, respectively.

In the next step, the developer has to run an IDL compiler that generates code for the interface and value type definitions. As the manager has to extract the current state and write it to the state store (cf. Fig. 3), there is a need for introspection. Therefore, methods for state introspection have to be generated for any CORBA-supported programming language that is not capable of native introspection. Thus, we provide a special IDL compiler for C++ and for Java, we use its native reflection capabilities.

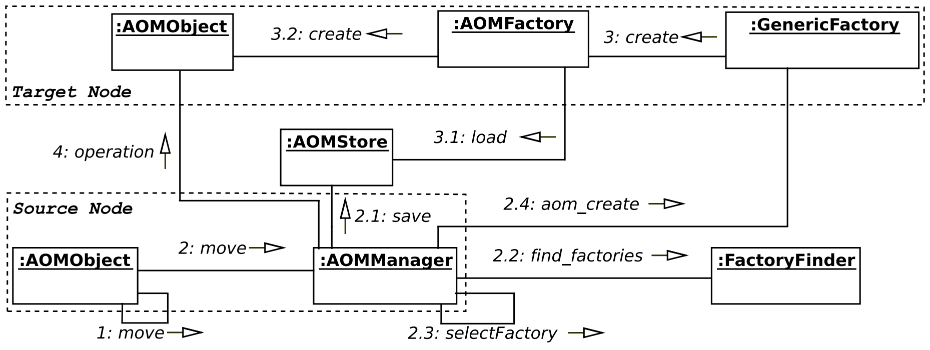

Fig. 4. Process of object migration

After having run the IDL compiler, the developer has to implement the actual value type realisations for all the programming languages and platforms that should be supported (supporting all facets of the mobile object). This also includes the implementation of the life-cycle operations. However, this effort is quite moderate because the whole migration functionality is provided by the `move()`-method of the provided `AOMManager` (cf. Figure 4). There, criteria (cf. Section 2) and information regarding the mobile agent facility have to be provided as parameters. This method just has to be called from within the `LifeCycleObject`'s `move()`-method. An implementation of the other life-cycle operations is simple as well [6]. For instantiation of the first object facet, the

[3] Implementation classes have the prefix AOM (adaptive object migration)

developer should use our factory. Therefore, the developer uses `aom_create()` of the `GenericFactory`. Then, the factory implicitly handles activation and sets the environment.

4.1 AOMManager

The `AOMManager` is responsible for triggering the adaptation process on the origin side (cf. manager, Fig. 3). First, it saves the active state of the value type to a state store (`AOMStore`). Therefore, the AOMManager uses the CORBA interface repository for obtaining the state information of a specific value type (i.e., the names of the state variables). For reading the state, native reflection is used for Java; for C++ the generated methods for introspection are used. This state is stored to the `AOMStore` referring the object identifier (cf. Fig. 4, 2.1).

Then, the `AOMManager` is responsible for searching for an appropriate target location for a pending migration. For this task, a known `FactoryFinder`, that is part of the AOM system, is queried on factories representing possible migration targets (within the scope of the `FactoryFinder`). These factories have to fulfil given `criteria`. Our provided `FactoryFinders` support `criteria` defining concrete locations (e.g., DNS-Name, IP-Address) and discrete locations specified by a required functionality (see Section 4.2).

The result of the query is a list of appropriate factories, which are able to instantiate the desired mobile object for realising the migration. Our `AOMManager` is able to select the "best" factory that is used for object migration. In our prototype implementation, we use the factory at the head of the list. As the `AOMManager` is a standard object we provide, user-defined policies can be implemented by providing a custom-build `FactorySelector` object. This object contains a custom `selectFactory()` method that can implement any user-defined selection strategy (cf. Fig. 4).

In the next step, the `AOMManager` invokes the remote method `aom_create()` at the `GenericFactory`. There, the fully-qualified name of the functionality and a reference to the `AOMStore` are transferred to the factory as parameters. In contrast to our recent Life Cycle Service implementation we do not transfer actual value types. This makes no sense as the `AOMManager` has low influence on the actual value type realization at the target. Therefore, the target has full access to the object's state by obtaining the reference to the `AOMStore`. The application developer decides if state should be stored in a local `AOMStore` by passing a boolean parameter (`attached`).

The `move()` method of the `AOMManager` has additional parameters for specifying the name and the parameters for an initialisation method that is invoked after a successful migration, which is especially useful for mobile agents acting autonomously. However, the developer should specify this method as a *one-way* method, which results in an asynchronous invocation.

If the mobile object has been created successfully at the factory, the original object is removed and the reference in the location service is updated to the new location (see [6]).

4.2 FactoryFinder

As already mentioned in Section 2, the `FactoryFinder` represents some kind of abstract location within the CORBA Life Cycle Service specification and provides methods for querying for appropriate factories acting as targets for object migration.

For supporting our AOM service, the `FactoryFinder`'s `find_factories()` method allows to search for registered factories residing at particular locations and providing particular functionalities, respectively (a list containing every appropriate factory is returned). For this purpose, `find_factories()` receives a CORBA Naming Service `NameComponent` sequence consisting of an <*identifier,kind*>-pair of strings. We specified two possible *kind*-values for our `FactoryFinder`. For a functionality, we specified "IFC" and for a location we specified "LOC". The particular *identifier* represents the functionality and the location, respectively. As `find_factories` receives a `NameComponent`-sequence, searching for specific functionalities at specific locations within the `FactoryFinder`'s repository is possible as well.

4.3 Generic Factory and AOMFactory

The conceptual transformer factory in Fig. 3 is actually represented by two entities in our implementation (`GenericFactory`, `AOMFactory`, cf. Fig. 4). The `GenericFactory` is responsible to create the actual `AOMFactory` for creating adapted objects. Therefore, the `GenericFactory` is able to use our Dynamic Loading Service (DLS, cf. Section 2) to load locally unavailable code on demand. This even enables instantiating previously unknown objects.

At the `AOMFactory`, the value type is created according to the transferred interface specification. If needed, the factory is even able to adapt the implementing value type according to the actual platform and criteria requirements. The active state is then loaded from the `AOMStore` into the adapted value type implementation. Therefore, the CORBA interface repository is queried for the state information of the actual value type, i.e., the names of the actual state attributes. This information is loaded from the state store and set in the value type by using native Java reflection and generated C++ introspection methods, respectively. After this process, the value type is activated, which enables accessing the object from remote clients.

If appropriate, i.e., if the variable `attached` is true, a local `AOMStore` is used for storing the complete state referring to a specific object identifier. The `AOMStore` has to be accessible as a CORBA initial reference (if there is no `AOMStore` as an initial reference, an error is reported to the application). Then, the complete state regarding a particular object identifier is loaded, stored locally and the state store variable in the actual value type is set to the local store. Later, instances can reference this state store remotely.

4.4 AOMStore

The `AOMStore` is an entity that is able to store the current state of a mobile object (cf. state store, Fig. 3). This entity is a plain CORBA object that can be located locally or remotely to the mobile object. The `AOMStore` contains the current state of an object related to an object identifier and offers two methods for loading and saving the state, respectively (cf. Fig. 5). The `load()` method takes the object identifier and the names of needed active state variables of the desired value type as a parameter and, thus, is able to return the appropriate part of the state.

```
interface AOMStore{
        void save (in OctetSeq oid, in AnySeq state);
        AnySeq load (in OctetSeq oid, in StringSeq values);
        AnySeq load_full (in OctetSeq oid);
};
```

Fig. 5. IDL interface of the `AOMStore`

The actual implementation of the `AOMStore` is left open to the developer. In our prototype, we realised a naive implementation that is based on Java HashMaps. However, an implementation that is realised by serialising the mobile object's state to external storage is possible as well (reducing memory load).

4.5 Accessing the AOM Object

In heterogeneous environments containing mobile objects, persistent object references are needed for clients to access the mobile object anytime. We realised these object references using a special location service [6]. Thus, object references point to the current location for the object's whole lifetime. However, adapting the functionality (interface) of a mobile object will probably result in remote call exceptions on the client side, as particular methods might not be specified within the new interface any more.

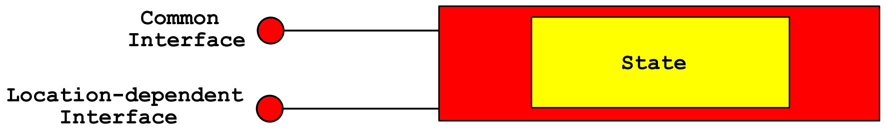

Fig. 6. Common and location-dependent interface of a mobile object

This problem can be solved by assuming that the mobile object has a common interface that does not change and a location-dependent interface that might change (cf. Fig. 6). This can be realised by a value type that has to support the common interface while optionally supporting other location-dependent interfaces. Thus, the location-dependent interface is just used for platform-internal purposes. Nevertheless, this interface is remotely accessible and can be used by clients (these have to handle exceptions if a special location-dependent interface is not existent).

As already mentioned in Section 1, our system is able to realise an agent that is able to adapt to the current location's needs. An adaptive mobile agent will have management methods for intervening into the agent's workflow (common interface). Besides, the agent will also offer methods that can be used on the current local node, e.g., for collecting data (location-dependent interface).

Beside pure migration support, these agents need a facility to restart their thread on the target side. These methods for initialising the agent are supported by the `AOMManager`. However, these methods should be declared *one-way* as this enables an asynchronous invocation. The `AOMManager` receives the method name and parameters and, then, is able to automatically invoke the method after a successful migration. Therefore, the CORBA dynamic invocation interface (DII) is used, as there is no way for the `AOMManager` to obtain the actual interface of the object after migration (the `AOMFactory` is able to adapt the object to the current context's needs).

5 Example Application

For outlining the development process and the necessary steps to implement an application using our adaptive mobile object concept, we present a simple distributed raytracing application.

Starting with the definition of a basic workflow, a developer has to identify the necessary interfaces and states of an adaptive mobile object. In context of our example application, first, the user defines the layout of a rendering scene, then submits the scene for processing and, in a final step, accesses the resulting frame. As indicated by the described workflow, we identified three different steps that can be modelled by an object supporting three facets providing different interfaces and state as displayed in Fig. 7.

We implemented a Java-based client application providing object facets for step one and three, represented by the `PerpareJob` and `JobResult` interface together with their associated value types `PrepareContainer` and `ResultContainer`. As these steps cover only get- and set-operations they can be performed on an arbitrary node running the application. The second step, however, performing the actual rendering process specified by the interface `RayTraceJob` and the associated value type `RaytraceContainer` requires slightly more resources and might be time consuming. Therefore, the rendering process should be done on a lightly loaded node having a powerful CPU for a highly optimised rendering implementation. The first criteria can be satisfied by a custom factory finder being able to locate such a node. The second one can be fulfilled by implementing the object facet in C++ using an appropriate library.

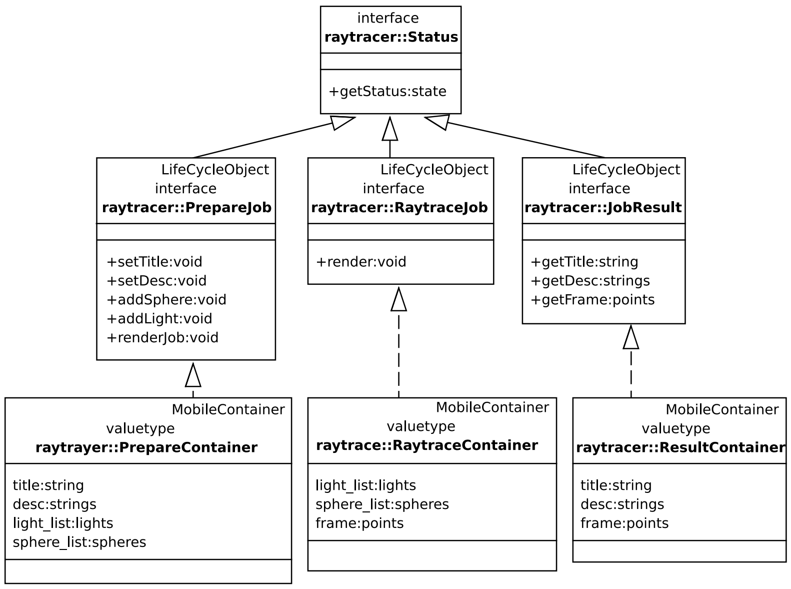

Fig. 7. Object facets of a basic distributed raytracing application

After specifying the scene via the set and add methods of the `PrepareJob` interface, a user calls `renderJob()` which turns the object into an agent, initiating the migration to the `RaytraceJob` interface that supports the rendering process (cf. Fig. 8, lines 2-6). Migration to an interface instead to a concrete location enables load balancing as our custom `FactoryFinder` selects an idle node. Along the lines to the interface change, the state is changed. While it is important for a user to have a title and description of a rendering scene, these information is negligible for the rendering process so these meta data is left out in the `RaytraceContainer` value type and, therefore, not migrated. Instead, it contains an additional field for the resulting frame. The migration process is finished by invoking the `render()` method as displayed in Fig. 8 (line 9). After successfull completion (line 10), the object at the source-location is implicitly removed by our service implementation. After processing the scene, the agent returns to its original location and changes the interface to provide the results. All three supported interfaces extend a simple `Status` interface providing information about the current workflow state. Thus, regardless of the current location, an application can monitor the execution process and, based on this information, narrow to one of the facets, e.g., to finally access the frame (`JobResult`).

For evaluation of the implementation, we compared the implementation of the presented example application using our former plain Life Cycle Service (LCS) with the implementation using the AOM infrastructure. Therefore, we implemented all supported facets and the LCS-based variant in Java and C++. The LCS object implementation supports all the interfaces by a union value type

```
public void renderJob() {
        NVP [] criteria = new NVP[1];
        Any ifcAny = orb.create_any();
        ifcAny. insert_string ("IDL:RayTraceJob:1.0");
         criteria [0] = new NVP("interface",ifcAny );
        Object obj = manager.move(criteria, this);

        RayTraceJob job = RayTraceHelper.narrow(obj);
        job.render();
}
```

Fig. 8. Realisation of interface-directed migration (Java)

comprising the complete state of all three facets. For ruling out network irregularities, the measurements were performed on a single AMD Opteron 2.2 GHz Linux server machine using JacORB 2.1 for Java and the Orbacus 4.30 for C++. We measured the roundtrip between the client application and a process acting as a rendering node without actually rendering the frame, as this would distort the measurements. Table 1 displays the results for the LCS and the AOM, respectively. The AOM implementation using only facets in C++ is almost 7 times faster than the Java implementation. One of the reasons is the usage of the interface repository for state transformation in Java (a query to the interface repository takes about 20ms). In C++ this is done by custom generated methods included in the value type code. As the comparison to the plain migration support provided by the LCS shows, flexibility provided by our adaptive mobile object infrastructure comes at some cost. However, we believe that benefits of adaptive object migration outweigh this performance penalty. Furthermore, our prototype has not been optimised for performance.

Migration	**Source**	**Target**	**Duration (ms)**
LCS	Java	Java	9.62 (± 0.48)
	Java	C++	5.08 (± 0.25)
	C++	C++	4.30 (± 0.21)
AOM	Java	Java	130.46 (± 6.52)
	Java	C++	82.70 (± 4.14)
	C++	C++	19.28 (± 0.96)

Table 1. Comparison of migration of a raytracing application using the standard CORBA Life Cycle Service and AOM

6 Related Work

In the past, many mobile agent systems have been developed. However, most of them only provide mechanisms for migration in homogeneous environments (e.g., MOA [10], Mole [11] or Aglets [12]). In general, these systems are Java-based and rely on the Java serialization mechanism. In this case, the state of an agent is tightly attached to the actual implementation. Thus, migrating a specific version to a new place with another implementation version will lead to serialization errors. Moreover, no adaptation of the mobile agent's state, functionality or code is supported at all.

Nevertheless, mobile agent systems for heterogeneous environments have been developed as well. In [3] a platform-neutral approach of agent migration is presented, based on transferring a blueprint instead of the code. This approach is realised by assuming that an agent consists of a set of different components. Then, a special *AgentFactory* is able to create an executable agent consisting of the right components from a received blueprint and related state. However, there is no support for the adaptation of state or functionality.

Takashio et al. present a mobile agent framework for supporting follow-me applications [1]. This framework allows mobile agents to adapt their code to the current context, i.e., applications are able to benefit from using high-performance implementations. However, this solution relies on Java and thus does not support heterogeneous programming-language environments. Additionally, there is no concept for adaptation of state and functionality, respectively.

Choy et al. describe a CORBA environment supporting mobile agents based on mobile CORBA objects [13]. Based on the Life Cycle Service a concept was developed, but apparently not implemented. Furthermore, the concept does not support adaptation of the mobile agent's state or code.

An Agent Transport Service (ATS) was specified in [14]. There, the CORBA Life Cycle Service was considered, but finally sorted out in order to support lightweight agents. As in our solution, all migration methods are offered entirely by the platform. In contrast to our approach, ATS does not provide support for adaptation of the agent's state or functionality.

Bellavista et al. propose the SOMA programming framework for mobile agents [4]. This work offers compliance to CORBA, and to the mobile agent standards MASIF and FIPA Thus, it achieves a good interoperability with many other mobile agent systems. However, SOMA neither supports adaptation of the mobile agent's interface nor of the agent's state.

Brandt et al. suggest reassembling agents from smaller subcomponents [15]. This allows exchanging environment-dependent implementations at runtime by selecting an appropriate implementation for a specific environment at runtime. Our approach is even better compared to this solution as our Dynamic Loading Service (DLS) [9] also allows selecting appropriate implementations for specific environments, and in contrast to Brandt et al. we also support changing the object's interface during runtime.

Almeida et al. propose a dynamic reconfiguration service for CORBA [16]. This service is able to upgrade objects without taking them offline by entailing operations for migration, replacement, addition and removal of objects. In contrast to our approach, developers have to implement methods for inspecting and modifying the state on their own and the dynamic reconfiguration service does not allow switching the object's interface or state adaptation.

The work of Garbinato et al. introduces Frugal Objects (FROBs) for mobile computing [17]. FROBs are objects within an event-based computing model, which provide adaptability according to their interface (which is represented by acceptable events) and their code. These objects support migration, but a drawback of this solution is the requirement of using a special non-intuitive FROB programming model that does not allow loops, forks or synchronization primitives.

The programming language Self [18] allows dynamic adaptation of local objects during runtime. Self is an object-oriented programming language based on the prototype concept, which allows manipulation of an object's methods and state at runtime. Instead of using inheritance for specialisation, an object developer copies an existent object and modifies the behaviour to the current needs. There is also a distributed Self variant called dSelf [19]. However, there is no support for heterogeneous object migration supporting other programming languages.

System	**Environment**	**Code**	**Functionality**	**State**
Self	Local	Static	Dynamic	Dynamic
dSelf	Homogeneous	Static	Dynamic	Dynamic
Aglets	Homogeneous	Dynamic	Static	Static
Takashio et al.	Homogeneous	Dynamic	Static	Static
FROBs	Homogeneous	Dynamic	Dynamic	Dynamic
Agent Factory etc.	Heterogeneous	Static	Static	Static
ATS	Heterogeneous	Static	Static	Static
SOMA	Heterogeneous	Static	Static	Static
Brandt et al.	Heterogeneous	Dynamic	Static	Dynamic
AOM	**Heterogeneous**	**Dynamic**	**Dynamic**	**Dynamic**

Table 2. Overview of capabilities of related migration approaches

Table 2 shows an overview of the capabilities of the presented related work. Our approach is the only one supporting dynamic migration of code, functionality and state in heterogeneous environments.

7 Conclusion and Future Work

We presented a novel approach of a dynamic adaptive object migration service for CORBA. Our service is build on top of the CORBA Life Cycle Service specification and, thus, provides compatibility to standard Life Cycle Service implementations. In contrast to previous implementations, we propose a separation of state, functionality and code. This enables providing an adaptive service that is capable of supporting heterogeneous platforms and programming languages.

CORBA is not per se a platform for Ubiquitous Computing as it is a fully-fledged middleware originally targeting at arbitrary distributed applications running on standard computers. This situation changes with the upcoming CORBA/e standard for embedded systems [20]. The new standard specifies a *compact profile* that supports CORBA value types. Thus, this enables a general transfer of our concept to embedded devices. However, as the CORBA/e standard does not support many dynamic features of standard CORBA we have to adapt our concept to these new requirements (e.g., CORBA/e does not support the interface repository and the dynamic invocation interface). However, using our approach in context of Ubiquitous Computing applications is already possible. There are CORBA implementations like TAO [21] that address small and also embedded devices even under real-time conditions.

Despite this fact, we will investigate to transfer our approach to other technologies, like, e.g., XML RPC and Web Services, for providing an infrastructure that needs fewer resources.

Our concept for adaptive object migration does not consider security so far. However, we enable applications handling security themselves by transferring a reference to actual state only. An integration of security mechanisms into our infrastructure is possible as well. Thus, for realising security, the remote state store will have to be trusted, allowing to secure the access to parts of the object's state. We will investigate the CORBA Security Service [22] for this purpose.

As the measurements showed, there is still the opportunity for performance optimisations. An obvious improvement would be the usage of custom code generation instead of native reflection for Java. However, we will also investigate to optimise the state transfer. For applications, which are able to statically define all facets before runtime, this can be realised by transferring a union value type representing the complete state of all facets instead of lists containing *Any* values. A static value type reduces serialisation costs and speeds up access to the state store.

Finally, we would like to offer extended support for developing applications basing on adaptive mobile objects. This includes a graphical UML-based modelling tool offering annotation support for the diverse facets of adaptive mobile objects and specialised code generators for multiple languages. Such a support should ease application development and help the developer to keep a consistent view across different facets, all supported platforms and multiple versions of adaptive mobile objects.

References

1. K. Takashio, G. Soeda, and H. Tokuda. A mobile agent framework for follow-me applications in ubiquitous computing environment. *ICDCSW*, 2001.
2. C. P. Kunze, S. Zaplata, and W. Lamersdorf. Mobile Process Description and Execution. In *6th Int. Conf. on Distr. App. and Interop. Sys.—DAIS*, 2006.

3. F.M.T. Brazier et al. Agent factory: generative migration of mobile agents in heterogeneous environments. In *ACM Symp. on Applied Comp.*, pages 101–106, Madrid, Spain, 2002.
4. P. Bellavista, A. Corradi, and C. Stefanelli. CORBA Solutions for Interoperability in Mobile Agent Environments. In *2nd Int. Symp. on Distr. Obj. and Appl.—DOA'00*, pages 283–292. IEEE, Sep 2000.
5. Object Management Group (OMG). Common object request broker architecture. OMG Document formal/2004-03-12, 2004.
6. R. Kapitza, H. Schmidt, and F. J. Hauck. Platform-Independent Object Migration in CORBA. In *OTM'05*, LNCS 3760, pages 900–917. Springer Verlag, Oct 2005.
7. Object Management Group (OMG). Life Cycle Service Specification. OMG Document formal/2002-09-01, 2002.
8. B. Liskov and S. Zilles. Programming with Abstract Data Types. In *ACM SIGPLAN Symp. on Very High Level Lang.*, pages 50–59. ACM Press, 1974.
9. R. Kapitza and F.J. Hauck. DLS: a CORBA service for dynamic loading of code. In *OTM Confed. Int. Conf.*, Sicily, Italy, 2003.
10. D.S. Milojicic, W. LaForge, and D. Chauhan. Mobile Objects and Agents (MOA). In *4th USENIX Conf. on OO Tech. and Sys.*, pages 179–194, 1998.
11. M. Strasser, J. Baumann, and F. Hohl. Mole: A Java based mobile agent system. *2nd ECOOP Works. on Mob. Obj. Sys.*, 1997.
12. D.B. Lange and M. Oshima. Programming and Deploying Java Mobile Agents with Aglets, 1998.
13. S. Choy, M. Breust, and T. Magedanz. A CORBA Environment Supporting Mobile Objects. Technical Report White Paper Draft Version 1, IKV++ GmbH, 1999.
14. C. A. Mendez and M. Mendes. Agent Migration Issues in CORBA Platforms. In *5th Int. Symp. on Autonom. Descentral. Sys.—ISADS'99*, Mar 1999.
15. R. Brandt, C. Hörtnagel, and H. Reiser. Dynamically Adaptable Mobile Agents for Scaleable Software and Service Management. *Journal of Comm. and Netw.*, 3(4):307–316, Dec 2001.
16. J. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In *3rd Int. Symp. on Distr. Obj. and Appl.—DOA'01*, pages 197–207. IEEE, Sept 2001.
17. B. Garbinato, R. Guerraoui, J. Hulaas, O. L. Madsen, M. Monod, and J. H. Spring. Mobile Computing with Frugal Objects . Technical report, EPFL I&C, 2005.
18. R. B. Smith and D. Ungar. Programming as an Experience: The Inspiration for Self. In *ECOOP '95 Conf.*, Aug 1995.
19. R. Tolksdorf and K. Knubben. Programming Distributed Systems with the Delegation-based Object-Oriented Language dSelf. In *ACM Symp. on Appl. Comp.—SAC'02*, 2002.
20. Object Management Group (OMG). Corba/e draft adopted specification. OMG Document ptc/06-05-01, 2006.
21. D. C. Schmidt. Real-time CORBA with TAO (The ACE ORB), May 2004.
22. Object Management Group (OMG). Security Service Specification. OMG Document formal/02-03-11, 2002.