Realizing the Leasing Concept in CORBA-Based Applications

Markus Aleksy
University of Mannheim
Department of Information Systems
D-68131 Mannheim, Germany
+49 621 181 1644
aleksy@uni-mannheim.de

Axel Korthaus
University of Mannheim
Department of Information Systems
D-68131 Mannheim, Germany
+49 621 181 1641
korthaus@uni-mannheim.de

Martin Schader
University of Mannheim
Department of Information Systems
D-68131 Mannheim, Germany
+49 621 181 1639
schader@uni-mannheim.de

ABSTRACT

Although in recent years the concept of leasing has become more and more popular in the field of distributed object-oriented systems, it has not yet been incorporated in the current specification of the widespread CORBA standard.

In this paper, we present an approach to the problem of realizing the leasing concept in a CORBA environment. We examine different leasing variants, such as exclusive or shared leasing, leasing with or without prolongation, etc., describe different ways to implement these variants in CORBA, e.g., based on a specialized object adapter, a runtime library or framework, or a CORBA service, and we discuss the advantages and disadvantages of each approach.

Furthermore, we show how existing CORBA services such as Naming, Trading, and Property Service can benefit from leasing facilities. Finally, we briefly present some architectural details of our own implementation of an exemplary CORBA Leasing Service.

Keywords
Leasing, CORBA, Services, Distributed Systems.

1. INTRODUCTION

Among the technologies currently available for the development of distributed object-oriented applications, the Common Object Request Broker Architecture (CORBA) [9] is one of the most popular and widespread approaches. Some of its outstanding strengths are the provision of independence from specific computer architectures, operating systems, programming languages, and vendors of ORB products. The last benefit was achieved for the first time in CORBA version 2.0, when unique object references, the so-called Interoperable Object References (IORs), and a standardized transmission protocol, the Internet Inter-ORB Protocol (IIOP), were introduced.

In order to abstract from the technical environment when developing CORBA applications, the Object Management Group (OMG) has specified the Interface Definition Language (IDL), which has to be used to describe the interfaces of CORBA objects that offer their services to clients. IDL is a purely declarative language, i.e., it is used to specify data types and interfaces containing attributes, operations, and exceptions only, not describing any internal algorithms of operations. The IDL is the foundation of CORBA’s programming language independence: in a separate step, the IDL-based interface definitions for CORBA application components have to be translated into a concrete programming language using an IDL compiler. Therefore, numerous language mappings from CORBA’s IDL to concrete programming languages such as Ada, C, C++, COBOL, Java, Lisp, PL/1, Python, and Smalltalk exist, and there are even non-standardized mappings to further programming languages such as Eiffel, Objective-C, and Perl, which are only available as part of proprietary ORB product bundles.

While the ORB is the basic component needed for mediating the communication between components in a distributed CORBA application, the OMG has also specified several system-level services, called CORBA services, to supplement the core functionality provided by the ORB with commonly required services for event processing, trading, persistence, and so on.

However, the concept of leasing, which, for example, plays an important role in the architecture of Jini [13] and is even supported by a dedicated specification issued by Sun Microsystems [12], has not found its way into the CORBA specification or the CORBA services yet. The potential benefits of the leasing concept, which are in no way limited to the Jini context, have been pointed out in the work of Jain and Kircher [3] who even speak of a “Leasing Pattern”. The authors work out the details of a best-practice design pattern including the formulation of the problem, the structure of the pattern, and possible variants of the pattern. The leasing concept is focused on a process of granting access to leased resources for a predefined period of time instead of letting clients access resources unrestrictedly. When the usage duration expires, several different alternative actions are possible, depending on the preferred policy, as will be explained in the following section. Since the foundations, motivations and the benefits of the leasing concepts have been described in detail in the cited literature, we will not repeat this information here but focus exclusively on potential ways to realize this concept in a CORBA-based environment.
The design and implementation of a CORBA Leasing Service, as described in this paper, is quite innovative, and currently we are not aware of any related work describing a similar approach.

2. THE LEASING CONCEPT

To get a better idea of what a leasing-based approach can achieve, think of the example of a software license, which can be regarded as a lease between the software and the user. Other examples could be electronic magazine subscriptions and web-based email accounts, where the subscription or usage right can be seen as leases that have to be renewed from time to time, e.g., if the duration of the subscription ends or if the email account has been inactive for a specific period of time [3].

There are many application fields for leasing approaches. Jain and Kircher [3] point out several general benefits of time-based leases, among which are simplicity of resource management, efficient resource usage, and enhanced system reliability. These advantages can be achieved because the user of a resource is freed from re-releasing resources explicitly, and the time-based leasing idea ensures that unused resources can be released automatically, so that they are not wasted and do not increase the system load unnecessarily.

Generally, the concept of leasing involves four different roles:

- **resource**
  A resource provides a specific service which can be requested by different service clients. The kind of service offered by the resource is irrelevant and should be viewed as being “opaque” from the leasing concept’s point of view.

- **resource claimant / holder**
  A resource claimant or holder, respectively, is a component that needs to use a resource. In order to be able to do so, the component must obtain a “lease”.

- **lease**
  A “lease” is the connecting link between a resource and a resource claimant. It represents permission for a resource claimant to use a resource for a specified duration. Moreover, it provides required functionality such as operations for releasing a resource prematurely (e.g., cancel) or, if the leasing variant allows it, to prolong the use of a resource (e.g., renew and renew_with_modified_time, respectively).

- **lessor / grantor**
  The lessor or grantor component is responsible for the management tasks resulting from the leasing concept, i.e., it manages the association of the resources with resource claimants by granting leases, thus allowing resource claimants to access resources for a specified period of time.

In the context of their examination, Jain and Kirchner [3] state the following requirements for an implementation of the leasing approach:

- **simplicity:**
  the management of resources for a holder should be simple by making it optional for the holder to explicitly release the resources that it no longer needs.

- **optimality:**
  the system load caused by unused resources must be minimized.

- **availability:**
  resources not used by a holder should be freed as soon as possible to make them available to new claimants.

2.1 Leasing Variants

Depending on the kind of leasing to be applied, the aspect of using resources for a predefined period of time can be emphasized more or less strongly.

Strict leasing requires that the resource be withdrawn from its holder immediately after the specified leasing duration is over, so that another claimant can use the resource. This means that the resource holder might need to subdivide its amount of workload because it will not be granted the full time required for the entire computation to be performed. Partitioning the algorithm can become very complex in this approach, so that it is much harder to implement, to understand, and to maintain than other approaches that are less strict. This strategy is primarily recommendable for very critical parts of an application.

A weaker variant of the leasing concept also allows for the renewal of the usage period, however limited by a total usage time not to be exceeded. In our opinion, this variant is to be preferred in most cases. The weakest leasing variant permits unrestricted prolongation of the usage time, thus, however, thwarting the goal of efficient resource usage.

Furthermore, we distinguish between exclusive leasing and shared leasing. Exclusive leasing means that one and only one resource claimant is assigned a specific resource during the usage period. This is the alternative of choice if the respective resource is stateful. Shared leasing, on the other hand, allows a resource to be assigned to more than one resource claimant at the same time, so that this variant is only suitable for stateless resources.

Another question to be considered refers to the authentication of the resource claimant. It has to be clarified whether a resource claimant has to prove its identity to the requested resource—which implies additional overhead—or whether one can rely on the assumption that all components in a distributed application adhere to the leasing concept and do not try to evade their usage restrictions.

3. IMPLEMENTATION OF THE LEASING CONCEPT IN A CORBA ENVIRONMENT

As already mentioned before, the current CORBA specification does not contain any information on how the leasing concept can be realized. Therefore, different CORBA-compliant implementation approaches are possible.

3.1 Using the POA Functionality

In CORBA, object adapters are used to mediate between the ORB and programming language servants. The Portable Object Adapter (POA) is the standard object adapter defined in the CORBA specification. It is responsible for creating object references, activating objects, and dispatching requests made on objects to their respective servants; thus it is involved in all aspects of an object’s life cycle.
Using the functionality of the POA represents the first alternative for the realization of leasing in the CORBA context. To this end, the POA specification would have to be extended. In that way, existing applications can continue to be used without any modifications, and all newly developed CORBA applications could benefit from the new functionality. However, this approach implies that ORB vendors would have to extend their products to support the enhanced POA. It is probable that this would result in major refusals by the ORB vendors, given that they already had to replace the underspecified Basic Object Adapter (BOA) of CORBA 2.0 with the POA only some years ago.

3.2 Implementation of an Additional Object Adapter

Another way to realize leasing functionality could be the implementation of an additional object adapter. As already mentioned before, the extension of POA functionality might lead to acceptance problems on the ORB vendors’ side. The introduction of an additional object adapter, which could be named “Leased Object Adapter” (LOA), would avoid this problem by insulating leasing functionality within an additional adapter. In this way, existing applications based on POA functionality could remain unchanged while new applications requiring leasing functionality could benefit from the new LOA. In order to minimize the effort of modifying existing applications to enrich them with leasing functionality, the LOA should fully support the POA functionality with all its policies, in addition to providing operations realizing the leasing concept.

As long as the new adapter is not part of the CORBA standard, the main disadvantage of this approach is its lack of portability. The reason is that developers would have to implement their own LOA, because there would be no standardized specification and standard-compliant implementations. Since an object adapter usually depends on details of the ORB implementation, it can only be built if these details are available, i.e., the developers of the object adapter would require access to the source code of the ORB. If CORBA applications are needed which use different ORBs at the same time, e.g., because different parts of the applications have been implemented with different programming languages or are executed on different platforms not supported by a single ORB product, the adapter has to be implemented more than once. This circumstance might lead to increased maintenance efforts for the application.

3.3 Design of a Leasing Framework

In order to be able to provide reusable leasing functionality for a multitude of different CORBA-based applications, the implementation could be realized in the form of a framework or a library, respectively. However, if more than one programming language is used for the implementation of a CORBA application, this approach is not very suitable. In this case, the library would have to be implemented for each programming language that might be used to access the library elements.

3.4 Realization of Leasing Functionality in the Form of a CORBA Service

In our opinion, the implementation of leasing functionality in the form of a CORBA service appears to be the best approach, because it follows the aims and objectives of the CORBA specification the best. As with the CORBA services specified in the standard, a Leasing Service would supplement the basic functionality of the ORB with additional, frequently used functionality. Because of its appearance as a CORBA service, the added functionality can be used by any application without any problem, independently of the programming language used for the implementation of the service. Moreover, applications can be written in any (other) programming language for which a CORBA language binding exists.

We consider this alternative the best solution with respect to compliance with the CORBA philosophy. Besides programming language independence, another resulting advantage is independence of the concrete ORB in use. Thus, acceptance problems are hardly to be expected and no changes to the CORBA standard are required. In the following sections, we will present an outline of the design of our Leasing Service.

4. A CORBA LEASING SERVICE

To enable CORBA-based applications to use their resources efficiently, we have developed a dedicated CORBA Leasing Service. In the following subsections, we are going to describe in detail the IDL interfaces the service provides, and we will explain the collaboration of the constituent components.

4.1 Design Considerations

The following considerations and goals guided our design:

- **Service approach**
  As mentioned before, a service-based approach is the best way to realize the idea of independence in CORBA. The functionality offered by the service can be used by all CORBA applications, no matter which programming language was used for their implementation. The only prerequisite is that a corresponding language binding to CORBA’s IDL exists.

- **Flexibility**
  For the Leasing Service to be applicable in a multitude of different contexts, we defined the notion of a resource to be as abstract as possible. Besides, different kinds of resource holders are supported.

- **Simple use**
  To ensure intuitive use of the functionality provided, we wanted the Leasing Service to fit the common human understanding of a leasing concept as good as possible.

- **Reuse of existing data types**
  In order to shorten the required learning phase for experienced CORBA developers, we adopted existing data types from other CORBA services such as Time::TimeT from the Time Service specification [10], and CosEventComm::PushConsumer from the Event Service specification [5]. Furthermore, we chose operation names that follow a similar scheme as operation names in the Event Service and the Licensing Service specifications [6] (e.g., obtain, etc.).

4.2 Leasing Service-Based Approach

Our CORBA Leasing Service mainly consists of the following interfaces that will be described in detail below and are illustrated in figure 1:

- **Resource**
4.2.1 Resource

In our approach, a Resource is considered opaque, i.e., the Leasing Service does not interpret the services offered by a Resource. Instead, the Leasing Service only expects a specification of the resource type and two operations needed to inform the Resource, which claimant requires its usage and when this usage will have to be terminated. With the help of operation start_use the resource is informed about the beginning of a new usage period. Thus, the resource has the opportunity to reset itself to an initial state. The invocation of operation stop_use indicates that the current usage period is over, so that the resource can release any other resources that have been allocated.

4.2.2 ResourceClaimant

A ResourceClaimant needs to use a Resource. To be able to do so, it must receive a Lease from the Lessor component. We distinguish three different kinds of ResourceClaimants: anonymous, observed, and notified. An anonymous resource claimant is the default; it is meant for clients not containing any CORBA objects. We left anonymous resource claimants out of figure 1, because they do not need to provide any specific callback operations.

The ObservedResourceClaimant receives a Lease, which not only has a limited validity but also observes the claimant. If the claimant terminates, the Lease will free the resource, even if the leasing time is not over. Since the lease must be able to observe the claimant, we have to provide a mechanism for this task. The easiest way to achieve this is to make use of the non_existent method of CORBA::Object, which can be queried to find out whether the respective object is still alive or not. Therefore, the ObservedResourceClaimant interface is just an alias for the CORBA::Object interface.

As opposed to the other two variants, the Lease automatically informs a NotifiedResourceClaimant when the leasing time expires. The respective Lease then automatically renews the leasing period for the resource once and only once in order to provide the NotifiedResourceClaimant with sufficient time for an adequate reaction. In order to be able to notify the claimant, we make use of the CORBA Event Service and define a NotifiedResourceClaimant simply as a CosEventComm::PushConsumer, so that the Lease can invoke the claimant's push operation for notification.

Therefore, a simple ResourceClaimant can be any kind of CORBA client and does not necessarily have to be a CORBA object itself, whereas ObservedResourceClaimants and

---

**Figure 1: Simplified illustration of the participating IDL interfaces**
NotifiedResourceClaimants need to be CORBA objects (more specifically, the latter even has to be a PushConsumer).

4.2.3 Lease
A Lease represents the link between a resource and a resource claimant. It provides a notion of time that can be associated with the availability of a resource. It provides several operations to fulfill its tasks. For example, with operations renew and renew_with_modified_time, respectively, a resource claimant can extend the resource usage time. Operation cancel can be invoked to free a resource prematurely, if it is no longer needed.

During its creation, a Lease generates a key, which is only valid for the granted leasing duration. A resource or a resource claimant using operation get_key can query this key. In that way, a resource can authenticate its users.

4.2.4 Lessor
The Lessor interface provides several different sets of operations. In the first category are operations used by resources. By calling operations register_resource or unregister_resource, for example, resources can make themselves known or unknown to the Lessor component. The second category refers to the claimants of resources. By invoking different list_operations, they can query the set of existing resource types, all resources of a specific type, or those resources of a specific type that are currently available. This last variant, i.e., listing available resources of a specific type, has to be used with care, because it can only provide a hint about which resources might be free. The reason is that when a resource claimant tries to allocate a resource that has been indicated to be available by the list operation, that very resource might have become allocated to another resource claimant in the meantime.

Furthermore, resource claimants can use several operations to obtain a Lease and therefore get access to a Resource. An anonymous resource claimant can request a Lease by calling Lessor operations obtain_lease_by_resource or obtain_lease_by_type. An ObservedResourceClaimant can demand a Lease by invoking operations obtain_lease_by_resource_with_client_observation or obtain_lease_by_type_with_client_observation. A NotifiedResourceClaimant uses operations obtain_lease_by_resource_withExpirationNotification or obtain_lease_by_type_withExpirationNotification in order to ask for a Lease.

As suggested by the operation names, these obtain operations can be divided into two groups, independently of the type of resource claimants. The first group contains operations for requesting a Lease for a specific resource. This can be interesting for clients that have already used a specific resource which meets their performance and quality requirements, so that they want to continue to use it. The second group contains operations that can be used by resource claimants to request a lease for a resource of a specific type. In this case, it makes no difference for the resource claimant which instance of this resource type will actually be assigned to it. If the operation is invoked several times and more than one resource of the specified type exists, it can be expected that the resource claimant will not be returned the same resource for each call.

5. UML SEQUENCE DIAGRAM SHOWING AN EXAMPLE SCENARIO
Figure 2 shows a UML Sequence Diagram corresponding to our implementation. It is an instance level diagram depicting the dynamic flow of messages in an example scenario for a system implementing the Leasing Service approach at runtime.

In the example, the claimant requires a specific instance of a resource type. The scenario begins with the registration of a resource with the Leasing Service. After that, a resource claimant demands the services of the resource at some later point in time after the registration process. If the claimant needs to find out which resource types are available, it can invoke list_available_types.

Should the desired type be available, the claimant can obtain all available instances by calling list_resources_by_type. According to its preferences, the claimant can now try to obtain a lease on its chosen instance by calling obtain_lease_by_resource.

Now, the Lessor creates a new instance of Lease (or takes one from an object pool), which then starts usage of the resource by calling start_use, provided that no other client has claimed the same resource in the meantime. Usually, managing a pool of Lease objects is more efficient than creating new objects every time they are needed and should therefore be provided by the service implementation.

Having obtained the lease, the claimant can now use the resource. In the example at hand, the time slot of the lease is extended by calling renew immediately before its expiration.

When the second time slot is over, the claimant does not renew the lease again and therefore the Lease frees the resource.

Further details about the technical foundation of our approach and example code can be found in [1].

6. APPLICATION OF THE LEASING CONCEPT IN CORBASERVICES
The ORB is the basic infrastructure component for the communication in distributed CORBA applications. As stated before, the OMG has specified a number of system-level services, called CORBA Services, in order to support the application developer. These services extend the basic functionality provided by ORBs with new functions commonly needed, such as event processing, trading of service offers, and concurrency control.

6.1 Naming Service
With its ability to localize distributed objects, the Naming Service [7] is the most fundamental service in CORBA. It was specified by the OMG as one of the first services and has now been implemented by nearly all ORB vendors. The task of the Naming Service is similar to that of a telephone directory. With the help of the Naming Service, arbitrary names can be assigned to CORBA objects (to references to these objects, to be precise). The association of a name with a context or an object reference, respectively, is called binding. A context again includes a set of bindings itself. A name is unique within its context and is mapped to exactly one other context or an object reference.
Here, the application of the leasing concept might be useful. The idea is that the Naming Service should not store its registered components forever, because this frequently results in holding object references and names that have become invalid, so that these resources continue to build up at the service. Thus, the service should store these references for a specified period of time only. If this leasing period is not renewed by the corresponding object, indicating that it is still alive and willing to offer its services, the Naming Service can then automatically delete the registered name and its corresponding object reference. Thus, "dangling" object references to CORBA objects that do not exist anymore can be avoided.

6.2 Trading Service

In open, heterogeneous, and distributed systems, name-based localization of a service provider sometimes is not a sufficient solution, because the service user often does not know the exact name of its communication partner. A better solution might be to provide search and selection functionality based on service properties and requirements. This approach is called "trading", and the CORBA Trading Service [11] is a popular realization of this concept.

The goal of trading is a market-like mediation between service providers and service clients. The users of the Trading Service can be grouped into two categories: service providers (exporters) and service users (importers). This classification is not exclusive or fixed, i.e., an exporter can as well act the role of an importer.

An exporter can export its service offer by providing a self-description containing access information, a service type, and several service attributes. An attribute represents a property of a service. It consists of an attribute name and an attribute type. Attributes can be static or dynamic. While static attributes serve to reflect the characteristics of a service, dynamic attributes reflect the service's current state. After registration with the Trading Service, an exporter can react to the requests of prospective clients.

An importer can obtain the required access information by specifying the desired service type and relevant attribute values. The

Figure 2: UML sequence diagram of an example scenario
mediating Trader might return no or any arbitrary number of object references of suitable exporters. To this end, it checks each registered service provider whether it complies with the required service type, and returns all suitable bindings. For its search, the Trader applies specific policies, which can be requested by the importer, so that the importer can influence, for example, the total number or the order of the service offers returned. The total number can become very high, so that a limitation of the potential results using specific policies, constraints, and preferences should always be considered.

In this context, the leasing concept can again be used to achieve some control over the storage of entries in the service. A Trading Service, which is based on the leasing concept, will find out if registered exporters are no longer alive or do not want to offer their services anymore, so that it can automatically remove the corresponding entries immediately.

In the literature, projects are described where distributed trader federations are examined (cf. [2]). In these federations, several trading services collaborate and exchange data. Here, the leasing concept could be applied in a similar way, however supervising not only the existence of registered service exporters but also of federated trader components.

6.3 Property Service

Using the Property Service [8], CORBA objects can be dynamically associated with additional “attributes” in the form of name-value pairs. In Mowbray [4], this technique is viewed as a self-contained design pattern, called “Dynamic Attribute Pattern”. Properties are not regular attributes as part of the object’s type, but are additional meta-information outside the static IDL type system.

A big disadvantage of the current Property Service specification is the lack of a mechanism for defining properties with time-limited validity, i.e., a restricted lifetime. As stated before, the main purpose of properties is to describe additional characteristics of objects. Some of these characteristics may be volatile and should therefore be deleted at a certain point in time determined by business rules. Another argument in favor of properties with limited lifetimes is of a more technical nature. Technical errors, poorly implemented applications, etc. can lead to situations where properties “survive” their corresponding objects. These properties, which have lost their associated objects, unnecessarily waste memory and have a degrading effect on the performance of the service and the application as a whole.

Using the leasing concept within the Property Service could solve this problem, since unused properties, i.e., properties whose leasing duration has not been renewed, could be automatically deleted by the service.

7. CONCLUSIONS

In this paper, we have discussed different ways to realize leasing functionality in the CORBA environment. The most flexible solution that best fits the CORBA philosophy is the implementation as a CORBA service. The Leasing Service we have designed and implemented closes a gap in the current specification of CORBA services by providing a new service offering leasing functionality to CORBA clients. Because of the design decision to choose a service-based approach, all CORBA applications can benefit from the result, independent of their programming language, the programming language of the service implementation, and the ORBs used—as long as those are CORBA-compliant. Our solution offers a great amount of flexibility, because of the far-reaching abstraction of the notion of resource and the support for different kinds of resource claimants. By using numerous elements already contained in other CORBA services, the learning phase for developers who are familiar with existing CORBA services could be kept quite short, so that the new functionality should be applicable quickly and easily.

8. REFERENCES