Leasing

Prashant Jain & Michael Kircher

{Prashant.Jain,Michael.Kircher}@mchp.siemens.de

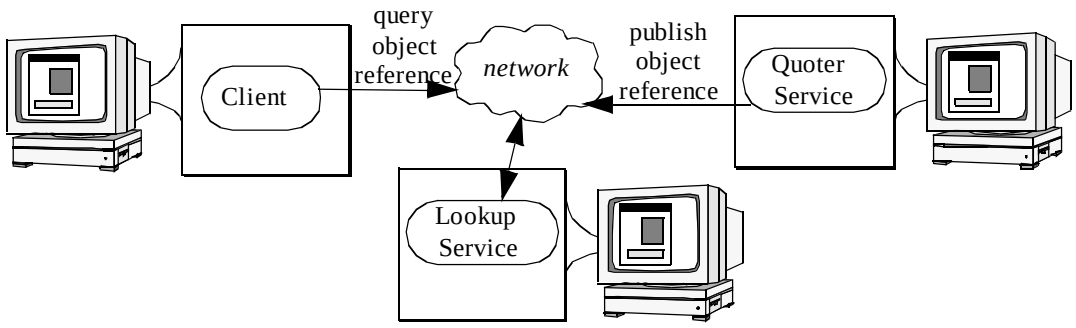
Siemens AG,

Munich, Germany

Leasing

The leasing pattern simplifies resource management by specifying how resource users can get access to a resource from a resource provider for a pre-defined period of time.

Example Consider a system consisting of several distributed objects implemented using CORBA [OMG]. To allow clients to access these distributed objects, the server containing these objects typically publishes the references of these objects in a Lookup Service [LOOKUP] such as a CORBA Name Service [NS]. Clients can then query the Lookup Service to obtain references to these objects. For example, consider a distributed Quoter service object registered with the CORBA Name Service. The Quoter service would provide stock quotes to any clients that connect to it. A client would typically query the Name Service, obtain a reference to the Quoter service, and then communicate directly with the Quoter service to obtain stock quotes.



Consider what would happen if the server containing the Quoter service were to crash and never come back up. The Quoter service would no longer be available but its reference would never get removed from the Name Service. This can create two problems. First, clients would still be able to obtain a reference to the Quoter service from the Name Service. However, the reference would be invalid and therefore any requests sent by the clients would typically result in an exception being thrown.

Secondly, lacking any explicit means to remove the invalid object reference, over a period of time unused resources such as invalid object references would continue to build up at the Lookup service.

Context Systems where resource usage needs to be controlled to allow timely release of unused resources.

Problem Highly robust and scalable systems must manage resources efficiently. A resource can be of many types including local as well as distributed services, database sessions and security tokens. In a typical use case, a user retrieves the interface of a resource provider and then asks the provider for one or more resources. Assuming the provider grants the resource, the user can then start using the resources. However, over a period of time, the user may no longer require some of these resources. Unless the user explicitly terminates its relationship with the provider and releases the resources, the unused resources would continue to be needlessly consumed. This in turn can have a degrading effect on performance of both the user and the provider. In addition, it can also affect resource availability for other users.

In systems where the resource user and the resource provider are distributed, it is also possible that over a period of time the provider machine may crash or that the provider may no longer offer some of its resources. Unless the user is explicitly informed about these resources becoming unavailable, the user may continue to hold invalid resources.

The net result of all of this is a build up of resources on the user side that may never get freed. One solution to this problem is to use some kind of a monitoring tool that could periodically check a user's resource usage as well as the state of the resources used by the user. The tool could then recommend to the user possible resources that can be freed. However, this solution is both tedious and error-prone. In addition, a monitoring tool may also hinder performance. To solve this problem in an effective and efficient manner requires resolving the following forces:

- *Simplicity*: the management of resources for a user should be simple by making it optional for the user to explicitly release the resources that it no longer needs.
- *Availability*: resources not used by a user should be freed as soon as possible to make them available to new users.
- *Optimality*: the system load caused by unused resources must be minimized.

- *Actuality*: a user should not use an obsolete version of a resource when a new version becomes available.

Solution Introduce a *lease* for every resource that is held by a user. A lease is granted by a *grantor* and is obtained by a *holder*. A lease grantor is typically the resource provider while a lease holder is typically the resource user.

A lease specifies a time duration for which the user can use the resource. Once the time duration elapses, the lease is said to have expired and the corresponding resource is freed from the user. While a lease is active, the lease holder can cancel the lease in which case the corresponding resource is also freed from the user. Before a lease expires, the lease *holder* can try to renew the lease from the lease *grantor*. If the lease is renewed, the corresponding resource continues to be available.

Structure The following participants form the structure of the Leasing pattern:

A *resource* provides some type of functionality or service.

A *lease* provides a notion of time that can be associated with the availability of a resource.

A *grantor* grants a lease on a resource.

A *holder* obtains a lease on a resource and then uses the resource.

The following CRC cards describe the responsibilities and collaborations of the participants.[1]

Class Resource	***Collaborator***
Responsibility • Provides application functionality	

Class Lease	***Collaborator*** • Grantor
Responsibility • Specifies a time period for which a resource is available • Informs the grantor on lease expiration	

Class Grantor	***Collaborator*** • Holder • Lease
Responsibility • Grants a lease on a resource to the holder	

Class Holder	***Collaborator*** • Resource • Lease • Grantor
Responsibility • Obtains and maintains a lease • Uses a resource • (Optionally) renews the lease	

Implementation There are four steps involved in implementing the Leasing pattern.

1 *Determine resources to associate leases with*. A lease should be associated with any resource whose availability is time-based. This includes resources that are short-lived, resources that are not used continuously by users, and resources that get updated frequently with newer versions.

2 *Determine lease creation policies*. A lease is created by the lease grantor for every resource used by a user. If a resource can be shared by multiple users, multiple leases will be created for the resource. A lease can be created by the lease grantor using a Factory [GHJV95]. Lease creation requires specifying the duration for which the lease is to

1. Class-Responsibility-Collaborators (CRC) cards [BRJ98] help to identify and specify objects or the components of an application in an informal way, especially in the early phases of software development. A CRC-card describes a component, an object, or a class of objects. The card consists of three fields that describe the name of the component, its responsibilities, and the names of other collaborating components.

be granted. The duration may depend upon the type of resource, the requested duration and the policies of the lease grantor. The lease requestor and the lease grantor may negotiate the duration for which the lease should be granted.

A user of a resource may want to pass the resource along with the associated lease to another user. The lease creation policies can be used to specify whether this is supported or not. If a resource along with its corresponding leases can be passed to other users, then the lease needs to provide operations allowing its ownership to be changed.

Once a lease has been created, the grantor needs to maintain a mapping between the lease and the corresponding resource. This allows the grantor to keep track of the duration of time for which the resource is being used and to determine which resources are still available for which new leases can be granted.

In addition, if the notification of users has to be supported a mapping of the lease to the corresponding user is necessary.

3 *Determine lease responsibility*. If a lease can be renewed, it needs to be determined who is responsible for renewing it. A lease may automatically renew itself or the renewal process may require re-negotiation between the grantor and the holder. A re-negotiation of the lease may result in new policies for the lease including the duration for which the lease is renewed.

4 *Determine lease expiration policies*: Once a lease expires and is not renewed, the resource assoicated with it needs to be released. This can be done automatically or may require some intervention on part of the user. Similarly, the lease grantor needs to remove the mapping between the lease, the resource and the user. Typically, the lease contains some kind of an Asynchronous Completion Token [POSA2] with information about the holder which it uses to allow proper cleanup in the grantor when the lease expires.

Example Resolved

Consider the example where a distributed Quoter service object needs to be available to CORBA clients. The server containing the Quoter service object would typically register the object with a CORBA Name Service. The Name Service would therefore be a resource provider while the resource will be the registration of the service object reference. The server containing the Quoter service object would be the user of the resource. The server and the Name Service would negotiate the lease details including

the duration for which the Quoter service object reference needs to be registered as well as policies regarding renewal of the lease. Once the negotiations are completed, the Name Service would register the Quoter service object reference with it and create a lease for the agreed-upon duration of time. The Name Service will be the lease grantor while the server will be the lease holder.

While the lease has not expired, the Name Service will keep the Quoter service object reference and make it available to any clients that request it. Once the lease expires, the server may need to explicitly renew the lease to indicate a continued interest in making the Quoter service object reference available to clients. If the server does not renew the lease, the Name Service will automatically remove the Quoter service object reference and release any additional resources associated with it.

The C++ code below shows how a server can register a Quoter service object reference with the Name Service. In the example below, the `LookupService` provides a wrapper around the Name Service, so that the standardized interface of the CORBA Name Service need not be changed. The `LookupService` serves as a lease grantor.

```
// First initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Create a Quoter service
Quoter_Impl *quoterServant = new Quoter_Impl;

// Get the CORBA object reference of it
Quoter_var quoter = quoterServant->_this();

// Get hold of the lookup service which is also
// the lease grantor
CORBA::Object_var obj = orb->
     resolve_initial_references("LookupService");

// Narrow the reference
Lookup::LookupService_var lookupService =
     Lookup::LookupService::_narrow(obj);

// Create an object specifying desired lease
// duration
TimeValue leasing_time (TimeValue::SECONDS, 10);

Lookup::Lease_var lease;

try {
     // Register the Quoter object reference with
```

```
     // the Lookup service
      lease = lookupService->bind(name,
                                  quoter,
                                  leasing_time);
} catch (Lookup::Negotiate)
{
     leasing_time = Lookup::TimeValue
                       (TimeValue::SECONDS, 5);
     // ... try again until you get a lease
}

// .. do other things

// Renew lease if we are still interested in
// publishing the object reference
lease->renew (TimeValue (TimeValue::SECONDS, 30));
```

The following code shows how the `LookupService` wraps the CORBA Name Service. After checking for an acceptable time period and creating a corresponding lease, the actual binding is delegated to the name service.

```
class LookupService_impl
: public POA_Lookup::LookupService {
public:
     // ...
     Lease_ptr bind (const CosNaming::Name &name,
                     CORBA::Object_ptr object,
                     const TimeValue &time) {
          if (this->time_is_acceptable (time)) {
               // Create a new lease
               Lease_Impl *lease_impl =
                    new Lease_Impl (time,
                              this->_this(), name);
               // Get the CORBA object reference
               LookupService::Lease_var lease =
                    lease_impl->_this ();

               // Add the new lease to our cache
               this->add_lease (lease, object);

               // Delegate
               nameService_->bind (name, object);
               return lease->_retn ();
          }
          else {
               // Reject the bind request
               throw (Lookup::Negotiate);
          }
     }
```

```
    void unbind (const CosNaming::Name &name) {
        // Delegate to the NS
        nameService_->unbind (name);
        // Do any other clean up required
        // including updating the cache
        // ... code omitted ...
    }

    TimeValue& renegotiate
             (LookupService::Lease_ptr lease,
              const TimeValue &time) {
        // Renegotiate lease. Code ommitted.
    }

    // ...
};
```

The following code shows how a lease could be implemented. Note that the Reactor pattern [POSA2] is used for the notion of timers. Therefore, the `Lease_impl` class implements not only the `Lease` interface but also the `Reactor::EventHandler` interface. This allows the lease to register itself with the reactor to receive timeouts.

```
class Lease_impl
: public POA_Lookup::Lease, Reactor::EventHandler
{
public:
    Lease_impl (const TimeValue &time,
        Lookup::LookupService_ptr lookupService,
        const CosNaming::Name &name)
    : time_ (time)
    , lookupService_(lookupService)
    , name_ (name)
    , reactor_ (Reactor::instance ())
    , valid_lease_ (TRUE) {};

    // Renew the lease for the given time
    void renew (const TimeValue &time) {
        // Renegotiate the lease with the grantor
        try {
            time_ = lookupService_->renegotiate
                        (this->_this (), time);
            // Reschedule timer with the reactor
            // for the duration of the lease
            reactor_->unregister_timer (this);
            reactor_->register_timer
                        (time, this);
        } catch (...) {
            // Error Handling
```

```
            }
        }

        // Callback method
        void on_timer_expire () {
            lookupService_->unbind (name_);
            valid_lease_ = FALSE;
        }

        // Method called by the lease holder to cancel
        // a lease.
        void cancel () {
            // Cancel timer with the reactor
            reactor_->unregister_timer (this);
            this->on_timer_expire ();
        }
        // ..
    };
```

When the lease duration expires, the reactor calls back the method `on_timer_expire` which in turn cleans up the appropriate resources.

Variants

Specific lease creation and expiration policies can yield various variants to the Leasing pattern. A lease may be created with the policy to automatically renew itself when its duration expires. In this case the lease maintains enough information about the holder and the grantor to automatically renew itself when its duration of time expires. Automatic renewals with short lease durations are preferrable over a single longer lease since each lease renewal gives the opportunity for the lease holder to update the resource it holds if the resource has changed. A further variation to this could be to limit the number of automatic renewals based on some negotiation between the lease grantor and the lease holder.

The renewal of a lease need not be done automatically by the lease or by the lease holder; instead, it can be done by a separate object. This can free the lease holder from the responsibility of renewing leases when they expire.

A lease may be created with no expiration. In this case, the holder must cancel the lease explicitly when it no longer needs the resource associated with the lease. This variant, however, loses many of the benefits of using the Leasing pattern but allows integration of legacy systems where the notion of leasing cannot be introduced easily.

Callbacks can be used to inform lease holders about expiring leases to give them a chance to renew those leases. This can help lease holders who do not want to or are not capable of determining when a lease will expire.

The Leasing pattern allows invalid resources such as object references to be released in a timely manner. The pattern can be extended using Invalidation [YBS] to allow invalid resources to be released explicitly. If a resource becomes invalid, the resource creator could send an invalidation signal to the lease grantor which could then propogate the signal to all the lease holders. The lease holders could then cancel the leases allowing the resource to be released. Note that Invalidation can result in additional complexity and dependencies between the resource creator, the lease grantor and the lease holders. It should therefore only be used when it is not sufficient to wait for the lease duration to expire and instead it is necessary to release resources as soon as they become invalid.

Known Uses

- **Jini**™ - Sun' s Jini™ technology makes extensive use of the Leasing pattern by using it in two ways. First, it couples each service with a lease object that specifies the duration of time for which a client can use that service. Once the lease expires and the client does not renew the lease, the service corresponding to the lease object is no longer available to the client. Second, it associates a lease object with each registration of a service with the Jini™ Lookup Service. If a lease expires and the corresponding service does not renew the lease, the service is removed from the Lookup Service.
- **Software licenses** - A software license can be regarded as a lease between the software and the user. A user may obtain a license for using a particular software. The license itself may be obtained, for example, from a license server and is usually for a set period of time. Once the period of time expires, the user must renew the license or else the software can no longer be used.
- **Magazine/Newspaper subscription** - A real world known use of the Leasing pattern is magazine and newspaper subscriptions. In this case, the subscription represents the lease which usually expires after a set period of time. The subscription must be renewed by the subscriber or else the subscription terminates. In some cases, the subscriber may set up automatic renewal, for example by providing bank account information or credit card information.

- **Web-based email accounts** - Many web-based email accounts, for example MSN Hotmail, accounts if not used for more than a certain period of time become inactive automatically. In this case, the duration of time can be regarded as a lease whose renewal requires use of the email account.

Consequences

There are several **benefits** of using the Leasing pattern:

Resource Management Simplicity: The Leasing pattern simplifies the management of resources for the user. It frees the user of a resource from the responsibility of releasing the resource explicitly. Once the lease on a resource expires and is not renewed by the user, the resource can be automatically released.

Effecient Resource Usage: A resource provider can control resource usage more effeciently through time-based leases. By bounding resource usage to a time-based lease, the resource provider can ensure that unused resources do not get wasted and are instead released as soon as possible allowing them to be granted to new users. This can lead to the overall system load caused by unused resources to be minimized.

Resource Update Simplicity: The Leasing pattern allows older versions of resources to be replaced with newer versions with relative ease. The resource provider can supply the resource user with a new version of a resource at the time of lease renewal.

Enhanced System Reliability: The Leasing pattern helps to increase system reliability by reducing the wastage of unused resources and by ensuring that resource users do not access invalid resources.

There are some **liabilities** of using the Leasing pattern:

Additional Overhead: The Leasing pattern requires an additional object in the form of a lease to be created for every resource that is granted by a provider to a user. Creating a pool of lease objects and re-using them with different resource allocations can help simplify this problem. In addition, on lease expiry, the lease grantor may need to send a notification to the lease adding additional overhead.

Additional Application logic: The Leasing pattern requires the application logic to support the concept of leases as the glue between the roles of resource providers and resource users. Application architects therefore need to design keeping in mind that resources are not unlimited and that they are not available all the time.

Timer Watchdog: The resource provider as well as the resource user need to be able to determine when a lease will expire. This requires support for some kind of a timer mechanism which may not be available in some legacy systems. If, however, the legacy systems are event-based applications then they can be made timer aware with very little overhead.

See Also The most common implementation of the Leasing pattern relies on event-based callbacks to signal lease expiration. An event-based application typically uses one or more event dispatchers or Reactors [POSA2], such as the Windows™ Message Queue, InterViews' Dispatcher [LC87], or the ACE Reactor [ACE]. Event dispatchers typically provide an interface to register event handlers such as timer handlers which get called on timer expiration.

In applications which do not contain an event loop, the Active Object [POSA2] design pattern can be used to substitue timer handling. The Active Object has its own thread of control and can either instrument the OS or run its event loop to signal leases via callbacks on timer expiration.

To make leasing transparent to the resource user the Proxy [GHJV95] design pattern can be employed. The resource proxy can handle lease renewals, policy negotiations, and lease cancellations that would otherwise normally be done by the user. CORBA Smart Proxies [SMP] provide the appropriate abstraction in the CORBA world while Smart Pointers are the pendant to this in traditional C++.

References

[ACE] http://www.cs.wustl.edu/~schmidt/ACE.html

[BRJ98] G. Booch, J. Rumbaugh, I. Jacobsen: *The Unified Modeling Language User Guide*, Addison-Wesley, 1998

[OMG] http://www.omg.org

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[JINI] http://www.sun.com/jini

[LC87] M.A. Linton, P.R. Calder: *The Design and Implementation of InterViews*, Proceedings of the USENIX C++ Workshop, November 1987

[LOOKUP] P. Jain and M. Kircher, “Lookup Pattern”,Submitted to European Pattern Language of Programs conference, July 5-9, 2000, Kloster Irsee, Germany

[NS] OMG, Interoperable Naming Service, Document orbos\98-10-11, 2000

[POSA2] D. Schmidt, F. Buschmann, H. Rohnert, M. Stal: *Pattern-Oriented Software Architecture—Patterns for Concurrent and Distributed Objects*, John Wiley and Sons, 2000

[SMP] http://www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/docs/Smart_Proxies.html

[YBS] H. Yu, L. Breslau, S. Shenker, “A Scalable Web Cache Consistency Architecture”, Computer Communication Review, ACM SIGCOMM, volume 29, number 4, October 1999