

---

# CORBA Session Management Guide C++

V6.3.14

# Table of Contents

Preface	4
Audience	4
Typographical conventions	4
Keying conventions	5
Using the Leasing Plug-In	6
The Leasing Framework	6
What is session management?	6
Features	6
Server side behavior	7
Client side behavior	7
Lease acquisition	8
Lease renewal	9
Client shutdown	9
A Sample Leasing Application	10
Location	10
The LeaseTest IDL module	10
Purpose	10
Client-server interaction	11
Using the Leasing Plug-In on the Client Side	11
Prerequisites	11
How to use the plug-in	11
Configuration variables	11
Configuring for co-located CORBA objects	12
Example configuration	12
Using the Leasing Plug-In on the Server Side	12
The IT_Leasing module	13
The LeaseCallback interface	14
The server lease agent interface	14
The Current interface	14

Implementing the server	14
Implement the LeaseCallback Interface	15
Use IT_Leasing::Current to Track Client Sessions	18
Advertise the Lease	25
Configure the Server-Side Plug-In	28
Leasing Plug-In Configuration Variables	29
Common Variables	29
List of variables	29
Server-Side Variables	30
List of Variables	30
Sample Leasing Plug-In Configuration	31
Configuration file extract	31
Leasing IDL Interfaces	33
The IT_Leasing IDL module	33
Glossary	42
Notices	46
Copyright	46
Trademarks	46
Examples	46
License agreement	46
Corporate information	47
Contacting Technical Support	47
Country and Toll-free telephone number	47

# Preface

---

This book describes the Orbix session management capability, which is based on the Orbix leasing plugin.

## Audience

---

This guide is aimed at developers of Orbix applications. Before reading this guide, you should be familiar with the Object Management Group IDL and the Java language.

## Typographical conventions

---

This guide uses the following typographical conventions:

### Constant width

Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class.

Constant width paragraphs represent code examples or information a system displays on the screen. For example:

```
#include <stdio.h>
```

<i>Italic</i>	<p>Italic words in normal text represent <i>emphasis</i> and <i>new terms</i>.</p> <p>Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:</p> <pre>% cd /users/*your_name* !!! note</pre> <p>Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with <i>italic</i> words or characters.</p>
---------------	--

## Keying conventions

---

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS or Windows command prompt.
... . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

# Using the Leasing Plug-In

---

*This chapter describes what the leasing plug-in does and how to use the leasing plug-in on the client-side and the server-side of your application.*

## The Leasing Framework

---

The leasing plug-in is an add-on feature for Orbix that manages server-side resources by detecting when client processes have ceased using a server. This is done using a leasing framework. When a client starts up, it can acquire a *lease* for a particular server, renewing it periodically. When the client terminates, it automatically releases the lease. If the client crashes, the server later detects that the lease has expired. In this manner, both graceful and ungraceful client process terminations are detected.

## What is session management?

---

It is a common requirement in many CORBA systems to know when a client process terminates, in order to clean up resources that are used only by that client. On the server side, session-based applications allocate resources to cater for client requests. To prevent servers from bloating, it is necessary to detect when clients are finished dealing with the server. CORBA does not provide a native solution to this problem.

## Features

---

The leasing framework has the following features:

- Zero impact on client application code.
- Zero impact on existing application IDL interfaces.
- Easy to implement.
- CORBA compliant.
- Completely configurable.

## Server side behavior

On the server side, the leasing framework operates as follows:

Stage	Description
1	When a server starts up, it automatically loads the leasing plug-in.
2	During initialization, the server advertises the lease, which causes a <code>LeaseCallback</code> object to be bound in the naming service.
3	Whenever the server exports object references (IORs), the plug-in automatically adds leasing information to the IOR in a CORBA-compliant manner.

## Client side behavior

On the client side, the leasing framework operates as follows:

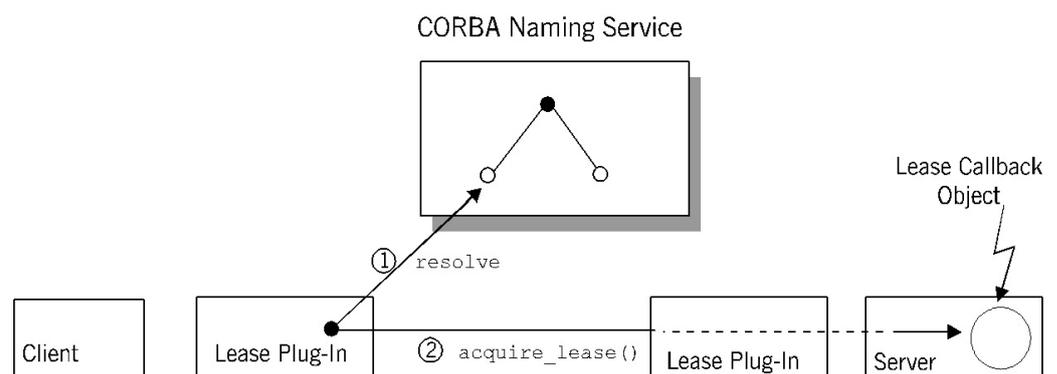
Stage	Description
1	When the client starts up, it automatically loads the leasing plug-in.
2	If the plug-in detects that the client is going to invoke on an object using an IOR containing leasing details, the plug-in automatically initiates a session with the target server by acquiring a lease.
3	The plug-in automatically renews the lease when needed.

Stage	Description
4	<p>Upon client shut down:</p> <ul style="list-style-type: none"> <li>- If the client shuts down gracefully, the plug-in automatically releases the lease with the server.</li> <li>- If the client crashes, the server-side plug-in later realizes that the client has not recently renewed the lease. The lease expires, allowing the server to clean up appropriately.</li> </ul>

## Lease acquisition

A client initiates a session by acquiring a lease from a leasing server, as shown in [Figure 1](#).

Figure 1: The Client Acquires a Lease



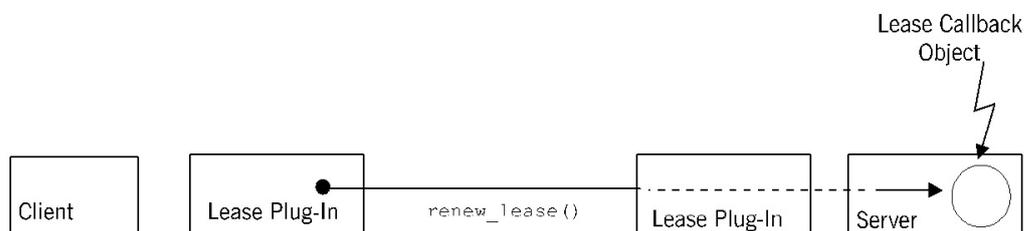
The client session is initiated by the leasing plug-in, as follows:

1. The client's leasing plug-in obtains an `IT_Leasing::LeaseCallback` object reference by resolving a name in the CORBA naming service.
2. The client's leasing plug-in initiates a session by calling `acquire_lease()` on the `LeaseCallback` object.

## Lease renewal

After acquiring a lease, the client renews the lease at regular intervals, as shown in [Figure 2](#)

Figure 2: The Client Renews the Lease

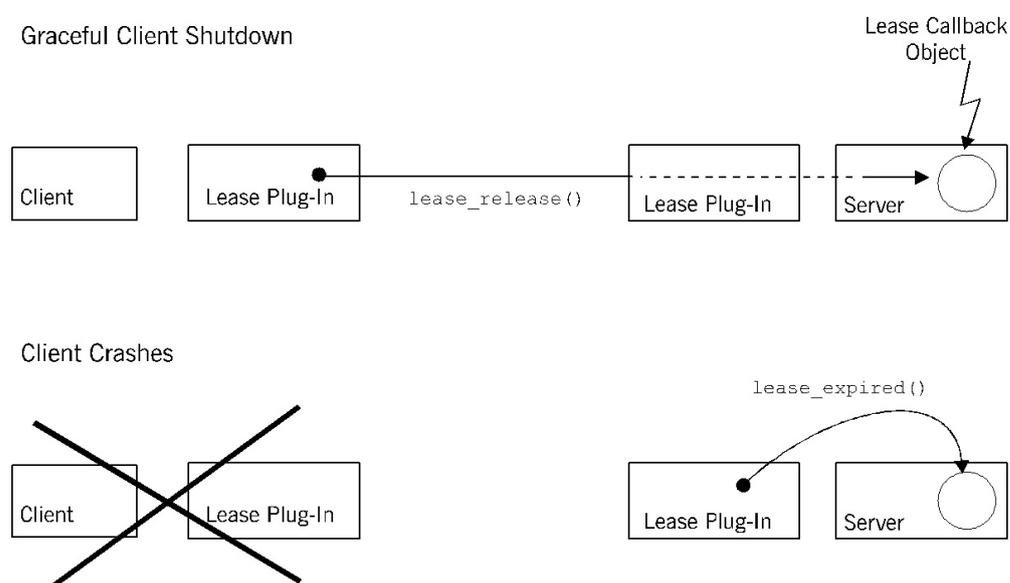


The period between lease renewals is specified by the `plugins:lease:lease_ping_time` configuration variable.

## Client shutdown

When the client shuts down, the lease is released as shown in [Figure 3](#)

Figure 3: The Lease is Released When the Client Shuts Down



The following shutdown scenarios can occur:

- *Graceful client shutdown*—if the client shuts down gracefully, the plug-in automatically calls `lease_release()` to end the session.

- *Client crashes*—if the client crashes, the server-side plug-in calls `lease_expired()` on the `LeaseCallback` object after a period of time specified by the `plugins:lease:lease_reap_time` configuration variable.

## A Sample Leasing Application

---

### Location

Source code and build instructions for a sample leasing application are located in the `asp/6.3/demos/corba/standard/session_management` directory of your Orbix installation.

### The LeaseTest IDL module

The sample leasing application is based on a server that supports a simple factory pattern for creating transient `Person` objects:

```
//IDL
module LeaseTest {
exception PersonAlreadyExists { };
interface Person {
string name();
};
interface PersonFactory {
Person create_person(in string name)
raises (PersonAlreadyExists);
};
};
```

### Purpose

The purpose of this example is to show that no matter how many clients create `Person` objects, and no matter how those client processes terminate, the server is notified when it can safely clean up the objects. Therefore, the server is able to keep its memory usage down.

## Client-server interaction

Clients interact with the `LeaseTest` server as follows:

Stage	Description
1	A client creates new <code>Person</code> objects by calling the <code>create_person()</code> operation, with unique <code>name</code> arguments for each <code>Person</code> .
2	When a client terminates, the <code>Person</code> objects it created no longer need to be held inside the server memory and are deleted.

## Using the Leasing Plug-In on the Client Side

### Prerequisites

The client plug-in makes periodic `resolve()` calls to the Naming Service during its lifetime. Therefore, your Orbix domain should have a properly configured locator, activator, and naming service ready before running a leasing client.

### How to use the plug-in

The only thing that needs to be changed in a client deployment that uses the leasing framework is its configuration. Specifically, the plug-in must be added to the list of ORB plug-ins and be configured to participate in bindings.

### Configuration variables

The following basic configuration variables are needed to configure and activate the client-side plug-in:

Table 1 Configuration Variables Used on the Client Side

Configuration Variable	Purpose
<code>plugins:lease:shlib_name</code>	Identifies the shared library that contains the plug-in code.
<code>plugins:lease:Classname</code>	Identifies the lease plug-in class name.

Configuration Variable	Purpose
<code>orb_plugins</code>	The ORB plug-in list is modified to ensure that the lease plug-in is automatically loaded when the client ORB is initialized.
<code>binding:client_binding_list</code>	The client binding list is modified to ensure that the plug-in can participate in request processing.

The complete set of leasing plug-in configuration variables is given in [Leasing Plug-In Configuration Variables](#).

## Configuring for co-located CORBA objects

In the `client_binding_list`, a binding description containing the `POA_Coloc` interceptor name *must* appear before the first binding description that contains a `LEASE` interceptor name. This is to ensure that a leasing application does not attempt to lease a co-located CORBA object.

## Example configuration

In an Orbix file-based configuration, the client-side plug-in might be configured as follows:

```
# Orbix Configuration File
plugins:lease:shlib_name = "it_lease";
plugins:lease:ClassName = "com.iona.corba.plugin.lease.LeasePlugIn";
orb_plugins = ["local_log_stream", "lease", "iiop_profile", "giop", "iiop"];
binding:client_binding_list = ["POA_Coloc", "LEASE+GIOP+IIOP", "GIOP+IIOP"];
```

## Using the Leasing Plug-In on the Server Side

## The IT\_Leasing module

Servers wishing to act as leasing servers interact with the plug-in to advertise leases. The interfaces used by leasing servers are declared in the `IT_Leasing` module, which is defined in the `leasing.idl` file:

```
//IDL
module IT_Leasing
{
  ...
  interface LeaseCallback
  {
    LeaseID acquire_lease()
    raises (CouldNotAcquireLease);
    void lease_expired(in LeaseID lease_id);
    void lease_released(in LeaseID lease_id);
    void renew_lease(in LeaseID lease_id)
    raises (LeaseHasExpired);
  };
  local interface ServerLeaseAgent
  {
    void advertise_lease(
    in LeaseCallback lease_callback
    ) raises (CouldNotAdvertiseLease);
    LeaseID manufacture_lease_id();
    void withdraw_lease();
    void lease_acquired(in LeaseID lease_id);
    void lease_released(in LeaseID lease_id);
  };
  local interface Current : CORBA::Current
  {
    exception NoContext {};
    LeaseID get_lease_id() raises (NoContext);
  };
  ...
};
```

The complete listing for the `IT_Leasing` module is in [Leasing IDL Interfaces](#).

## The LeaseCallback interface

Your server must provide an implementation of the `IT_Leasing::LeaseCallback` interface to receive notifications of lease-related events from the leasing plug-in. For example, when leases expire, the plug-in calls `IT_Leasing::LeaseCallback::lease_expired()`.

## The server lease agent interface

The implementation of the `ServerLeaseAgent` interface is provided by the leasing plug-in. Your server communicates with the leasing plug-in by calling the operations defined on this interface. For example, the server can initialize the leasing plug-in by calling `IT_Leasing::ServerLeaseAgent::advertise_lease()`.

## The Current interface

For a leasing server to react correctly to the *ending* of a lease, it must know which resources are relevant to that lease. In other words, the server must maintain an association between the resources that it has created and the clients that are currently using them.

This problem is solved as follows. When your server needs to figure out which leasing client invoked a particular operation, you can extract lease information from an object of `IT_Leasing::Current` type, which is derived from `CORBA::Current`, an interface specifically used for retrieving meta-information about CORBA invocations. Once the `IT_Leasing::Current` object is obtained, you can call `get_lease_id()` on it to find the lease ID relevant to that call.

If the call is made from a non-leasing client (or a non-Orbix client), the `IT_Leasing::Current::NoContext` user exception is thrown.

## Implementing the server

To use the plug-in on the server side, perform the following steps:

Step	Action
1	<a href="#">Implement the LeaseCallback Interface.</a>
2	<a href="#">Use IT_Leasing::Current to Track Client Sessions.</a>
3	<a href="#">Advertise the Lease.</a>

Step	Action
4	<a href="#">Configure the Server-Side Plug-In.</a>

## Implement the LeaseCallback Interface

You must implement the `LeaseCallback` interface to receive notification of leasing events from the plug-in.

The following example shows a code extract from the `LeaseTest` demonstration, where the `LeaseCallback` interface is implemented by the `LeaseCallbackImpl` class.

### Object instances

The following two object instances are used by the `LeaseCallbackImpl` class:

Table 2 Object Instances Used in the LeaseCallbackImpl Class

Object Instance	Description
<code>m_lease_obj</code>	An <code>IT_Leasing::ServerLeaseAgent</code> object reference. This object is used to communicate with the leasing plug-in.
<code>m_factory</code>	A reference to a <code>PersonFactoryImpl</code> object. This object is used to create new instances of <code>Person</code> CORBA objects.

### Implementation code

The `IT_Leasing::LeaseCallback` interface is implemented by the `LeaseCallbackImpl` Java class, as shown in [Example 1](#).

## Example1 The LeaseCallbackImpl Class

```

//Java
package demos.session_management.LeaseTest;
//--JDK Imports--
import java.io.*;
//--IONAImports--
import demos.session_management.LeaseTest.*;
import com.ionacorba.IT_Lease_Component.*;
import com.ionacorba.IT_Lease_Logging.*;
import com.ionacorba.IT_Leasing.*;
import com.ionacorba.plugin.*;
import
com.ionacorba.util.SystemExceptionDisplayHelper;

class LeaseCallbackImpl extends LeaseCallbackPOA
{
private PersonFactoryImpl m_factory = null;
private ServerLeaseAgent m_lease_obj = null;
// Constructor (not shown)
...

// IDL operations

[1](#implementation-
code) public String acquire_lease()
{
// We could throw CouldNotAcquireLease here if we
// wanted to refuse the lease
if (m_lease_obj == null)
{
System.err.println(
"ERROR: The Lease callback object has not been set
correctly.");
System.exit(1);
}
String new_lease =
m_lease_obj.manufacture_lease_id();
m_lease_obj.lease_acquired(new_lease);
return new_lease;
}

```

[2](#implementation-code)	<pre>public void lease_expired(String lease_id) {     m_factory.owner_has_gone_away(lease_id); }</pre>
[3](#implementation-code)	<pre>public void lease_released(String lease_id) {     m_lease_obj.lease_released(lease_id);     m_factory.owner_has_gone_away(lease_id); }</pre>
[4](#implementation-code)	<pre>public void renew_lease(String lease_id) {     // Nothing to do, since the plugin has already     // intercepted     // this request and knows that the lease has been     // renewed. } }</pre>

The code can be explained as follows:

1. The `LeaseCallbackImpl.acquire_lease()` method is called by client lease plug-ins when they need to acquire a lease with your server. The sample implementation asks the lease plug-in for a new unique lease ID, and then informs the plug-in that it has accepted the lease acquisition request by calling `lease_acquired()` on the `ServerLeaseAgent` object. You could also create the lease ID yourself—however, you are then required to ensure its uniqueness within the server process.
2. The `LeaseCallbackImpl.lease_expired()` method is called by the plug-in when a particular lease has expired—that is, if the lease has not been renewed within the configured reap time (see [Leasing Plug-In Configuration Variables](#)). This can occur if the client crashes or if the network link is lost between the client and the server.

The sample implementation informs the `Person` factory that a particular owner of `Person` objects has disappeared, by calling `owner_has_gone_away()`. The `Person` factory is then free to remove any `Person` objects belonging to that client. The sample `PersonFactory` removes the `Person` objects from a hash table, which allows the garbage collector to free the associated memory. Alternatively, a server could *evict* the transient objects by persisting their data before removing them from the hash table.

3. The `LeaseCallbackImpl.lease_released()` method is called by client lease plug-ins when the client shuts down gracefully. The implementation of this method is typically almost identical to the implementation of `lease_expired()`, because they are both caused by client terminations. The sample code delegates to the `PersonFactory` servant, informing it that a particular client has shut down.

There is one important difference between `lease_released()` and `lease_expired()`, however. When `lease_released()` is invoked, you should inform the plug-in of the event, so that it stops managing that particular lease and checking for its expiration. Do this by calling `ServerLeaseAgent::lease_released()`, as in the example code.

4. The `LeaseCallbackImpl.renew_lease()` method is the ping method that the client plug-ins call periodically to renew their leases. You can leave this function body empty. By virtue of the call reaching this point, it has already been intercepted and examined by the server-side plug-in. During the interception, the lease is timestamped with the current time as its *last renewed time*. You might want to perform some logging here.

## Use IT\_Leasing::Current to Track Client Sessions

---

The server has to track the resources associated with each client and this is done with the help of the `IT_Leasing::Current` interface. In the `LeaseTest` example, the associated resources are `Person` objects. Whenever a `Person` object is created (using the `LeaseTest::PersonFactory` interface) the server associates the new `Person` object with the current client session.

The current client session is identified by the current lease ID, which is obtained from the `IT_Leasing::Current` interface.

### Implementation code

The `LeaseTest::PersonFactory` interface is implemented by the `PersonFactoryImpl` Java class as shown in [Example 2](#).

## Example2 The PersonFactoryImpl Class

```

//Java
package demos.session_management.LeaseTest;
/--JDK Imports--
import java.io.*;
import java.util.*;
/--OMG Imports--
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POAPackage.*;
/--IONAImports--
import
com.iona.corba.util.SystemExceptionDisplayHelper;
import com.iona.corba.IT_Leasing.*;
import com.iona.corba.IT_Leasing.CurrentPackage.*;
class PersonFactoryImpl extends PersonFactoryPOA
{
// The set of People that the Factory is currently
managing
private Hashtable m_people = new Hashtable();
private ORB m_orb;
private POA m_poa;
// Constructor
... // (not shown)

public Person create_person(String name)
throws PersonAlreadyExists
{
Person result = null;
try
{
System.out.println("LeaseTest.create_person("+name+")"
);
}
}
}

[1] String owner = "<unknown>";
(#implementation-
code)

```

	<pre>try {</pre>
[2] (#implementation-code)	<pre>org.omg.CORBA.Object objref = m_orb.resolve_initial_references("LeaseCurrent");</pre>
	<pre>if (objref != null) { com.iona.corba.IT_Leasing.Current current = com.iona.corba.IT_Leasing.CurrentHelper.narrow( objref );</pre>

```

[3]      owner = current.get_lease_id();
(#implementation-
code)    }
        }
        catch (NoContext nc)
        {
        System.err.println(
        "Couldn't find the relevant ServiceContext data.");
        }
        catch (InvalidName in)
        {
        System.err.println("Caught InvalidName exception.");
        }
        catch (SystemException se)
        {
        System.err.println("Unknown exception"
        + SystemExceptionDisplayHelper.toString(se));
        }
        // Create a new Person servant and activate it
        PersonImpl newPersonServant;
        byte[] oid;
        org.omg.CORBA.Object tmp_ref = null;
        synchronized (this)
        {
        // check for Person existence within this process
        if (person_is_alive(name))
        {
        System.err.println("Person already exists!");
        throw new PersonAlreadyExists();
        }
        else
        {
        // Person does not exist, so it is created and
        // stored with the others, indexed by its name
        [4]      newPersonServant = new PersonImpl(name, owner);
(#implementation-
code)

```

```

try
{
oid = m_poa.activate_object(newPersonServant);
tmp_ref = m_poa.id_to_reference(oid);
}
catch (ServantAlreadyActive sae)
{
System.err.println(
"Unexpected ServantAlreadyActive exception.");
}
catch (WrongPolicy wp)
{
System.err.println(
"Unexpected WrongPolicy exception.");
}
catch (ObjectNotActive one)
{
System.err.println(
"Unexpected ObjectNotActive exception.");
}
result = PersonHelper.narrow(tmp_ref);
if (result == null)
{
System.err.println("Person is null error");
System.exit(1);
}
// store the new servant with the others
String temp_string = new String(name);

```

```

[5] m_people.put(temp_string, newPersonServant);
(#implementation- System.out.println("Created: " + name);
code) dump_people_to_screen();
}
}
}

```

```

catch (PersonAlreadyExists pae)
{
throw pae;
}

```

	<pre> catch (SystemException se) { System.err.println("Unexpected system exception." + SystemExceptionDisplayHelper.toString(se)); } </pre>
[6] (#implementation-code)	<pre> return result; } </pre>
[7] (#implementation-code)	<pre> void owner_has_gone_away(String owner) { // Iterate through the people map and evict any people // who were created by 'owner'. // Hashtable tmp_table = new Hashtable(); tmp_table.putAll(m_people); Set the_set = tmp_table.keySet(); String this_owner = null; </pre>
	<pre> if (!the_set.isEmpty()) { Iterator the_iter = the_set.iterator(); </pre>
	<pre> do { String key = (String)the_iter.next(); PersonImpl the_person = (PersonImpl)tmp_table.get(key); this_owner = the_person.owner(); </pre>
	<pre> // value may == null if this has already been evicted // while we are iterating through the list. if (owner.equals(this_owner)) { </pre>

	<pre> try { // deactivate the servant before deleting it byte[] oid = m_poa.servant_to_id(the_person); // deactivate the servant with the corresponding // id on the POA </pre>
[8] (#implementation- code)	<pre> m_poa.deactivate_object(oid); } </pre>
	<pre> catch(ObjectNotActive one) { System.err.println( "ERROR: Unexpected ObjectNotActive exception."); } </pre>
	<pre> catch(WrongPolicy wp) { System.err.println( "ERROR: Unexpected WrongPolicy exception."); } </pre>
	<pre> catch(ServantNotActive sna) { System.err.println( "ERROR: Unexpected ServantNotActive exception."); } </pre>
[9] (#implementation- code)	<pre> m_people.remove(key); } } while(the_iter.hasNext()); } dump_people_to_screen(); } ... } </pre>

The code can be explained as follows:

1. If the factory cannot figure out the relevant lease ID, it assigns a default ID of `<unknown>` as the owner of the object. This happens if a non-leasing client (either a non-Orbix client or an Orbix client that did not load the plug-in) invokes the factory.
2. The factory checks to see if it can contact the `LeaseCurrent` object.
3. If a reference to a `LeaseCurrent` object can be obtained, the `get_lease_id()` method is called to get the lease ID (of string type) for this invocation.
4. A new `Person` object is created and activated. The `result` variable is set equal to the corresponding `Person` object reference.
5. The factory stores the new `Person` object in its own internal table of `Person` objects, `m_people`, using the lease ID, `temp_string`, as a key.
6. The `Person` object reference, `result`, is returned to the calling code.
7. The `owner_has_gone_away()` method is called by `LeaseCallback::lease_expired()` or `LeaseCallback::lease_released()` to clean up the resources (`Person` objects) associated with a client session identified by the `owner` string. The code iterates over all of the entries in the `m_person` table, searching for entries associated with the `owner` session.
8. Before removing a `Person` object from the hash table, the corresponding servant must be deactivated by calling `PortableServer.POA.deactivate_object()`.
9. The servant object is removed from the `m_people` hash table in this line of code. This allows the Java garbage collector to free the associated memory.

## Advertise the Lease

---

### Prerequisites

Advertising the lease causes the `LeaseCallback` object reference to be bound into the naming service. Therefore, you must have your Orbix locator, node daemon, and naming service properly configured and ready to run.

### Where to advertise

Lease advertisement is an initialization step that is performed in the server `main()` method. This should be done before the server starts to process incoming CORBA requests (that is, before the server calls `ORB.run()` or `ORB.perform_work()`).

### Implementation code

The code shown in [Example 3](#) should be added to your server's `main()` method to advertise the lease:

## Example3 Advertising the Lease in the main() Method

```

//Java
package demos.session_management.LeaseTest;
// Imports (not shown)
...
class Server
{
...
public static void main(String args[])
{
...
ServerLeaseAgent leaseObj = null;
...
// Contact the Lease Plugin

```

```

try
{

```

```

[1](#implementation-
code)

```

```

tmp_ref = orb.resolve_initial_references(
"IT_ServerLeaseAgent"
);
leaseObj =
ServerLeaseAgentHelper.narrow(tmp_ref);
}

```

```

catch (InvalidName in)
{
// Process the exception ...
}

```

```

catch (SystemException se)
{
// Process the exception ...
}
...

```

```

        // Assume that we have already created and
        activated a
        // LeaseCallback servant and created a reference
        for it
        // called the_LeaseCallbackObject.
        ...

        // advertise a lease on the lease plugin

        try

        {

[2](#implementation-      leaseObj.advertise_lease(the_LeaseCallbackObject
code)                    );

        }

        catch (CouldNotAdvertiseLease cna)
        {
        // Process the exception ...
        }
        catch (DuplicateServerID dsid)
        {
        // Process the exception ...
        }
        catch (SystemException se)
        {
        // Process the exception ...
        }
        ...
        }
        }

```

The code can be explained as follows:

1. The server obtains an initial reference to a `ServerLeaseAgent` object, which is created by the leasing plug-in.
2. The leasing plug-in is initialized by calling `advertise_lease()` on the `ServerLeaseAgent` object. The `advertise_lease()` operation takes a single parameter, `the_LeaseCallbackObject`, which causes the `LeaseCallback` object to be registered with the plug-in.

## Configure the Server-Side Plug-In

Server-side configuration variables are used to initialize the server-side plug-in and to customize the behavior of the leasing plug-in. Some of these configuration variables are communicated to clients by inserting the information into IORs generated by the server.

### Configuration variables

In addition to the client-side configuration variables, the following basic configuration variables are needed to configure the server-side plug-in:

Table 3 Configuration Variables Used on the Client Side

Configuration Variable	Purpose
<code>binding:server_binding_list</code>	The server binding list is modified, instructing the ORB to insert <code>LEASE</code> interceptors into server-side bindings.
<code>plugins:lease:lease_name_to_advertise</code>	The name under which the <code>LeaseCallback</code> object is bound in the naming service. This name must be unique per server.
<code>plugins:lease:lease_ping_time</code>	The time interval (in milliseconds) between successive ping messages sent by client-side plug-ins to renew the lease.
<code>plugins:lease:lease_reap_time</code>	If a particular client's lease is not pinged within <code>lease_reap_time</code> , the server resources associated with the client are released.

The complete set of leasing plug-in configuration variables is given in [Leasing Plug-In Configuration Variables](#).

### Example configuration

For a complete example of a client-side and server-side configuration, see [Sample Leasing Plug-In Configuration](#).

# Leasing Plug-In Configuration Variables

---

The following list describes the leasing plug-in configuration variables and their allowed values, ranges, and defaults.

## Common Variables

---

### List of variables

---

The following configuration variables apply to both clients and servers:

#### **event\_log:filters**

Specifies a list of logging filters. You can configure the plug-in to write to a log stream by appending the plug-in log stream to the list of filters (see the *Orbix Administrator's Guide* for more information on log stream configuration). The plug-in's log stream object is `IT_LEASE`. For example, to get full diagnostic output from the plug-in, set the variable `event_log:filters` equal to `["IT_LEASE=*"]`.

#### **plugins:lease:lease\_ns\_context**

Identifies the naming service `NamingContext` where the leasing plug-in registers the `LeaseCallback` object. The name should be a valid `NamingContext` id (see the CORBA Naming Service specification). Since both leasing clients and leasing servers use this value, it should be set to the same value across your entire domain. The default is `IT_Leases`.

#### **plugins:lease:ClassName**

Identifies the entry point for the Java leasing plug-in code. The `ClassName` variable should be set to the leasing plug-in class name, which is `com.iona.corba.plugin.lease.LeasePlugIn`.

# Server-Side Variables

---

## List of Variables

The following configuration variables apply only to servers:

### **plugins:lease:allow\_advertisement\_overwrites**

Determines whether the server can re-advertise the same lease when it comes back up after a crash or disorderly shutdown. Internally, the plug-in uses `NamingContext::rebind()` if set to `true`, or `NamingContext::bind()` if set to `false`, when binding the `LeaseCallback` object in the naming service.

The default is `false`, but in a real deployment scenario the recommended setting is `true`.

### **plugins:lease:lease\_name\_to\_advertise**

Determines the lease name used when registering the `LeaseCallback` object in the naming service. This name should be configured to be unique among all your leasing servers. The name should be a valid `NamingContext` id (see the CORBA naming service specification). The default value is `default_lease_name`.

### **plugins:lease:lease\_ping\_time**

Determines the value inserted into `TAG_IONA_LEASE` IOR components for the lease ping time. Leasing clients using that IOR automatically renew the lease by pinging every `N` ms, where `N` is the value specified in this variable. The default value is 900,000 ms (15 minutes). Legal values are unsigned longs  $> 1$ . In addition, if the ping time is specified to be greater than the reap time, `lease_reap_time`, it is automatically changed to half the reap time.

### **plugins:lease:lease\_reap\_time**

Determines how often the server-side plug-in checks whether leases have expired. The value is specified in ms. If a particular lease has not been renewed (pinged) by its client in this amount of time, the lease expires. Legal values are unsigned longs  $> 2$ . The default value is 1,800,000 ms (30 minutes).

# Sample Leasing Plug-In Configuration

---

*This appendix shows the leasing plug-in configuration used in the session management demonstration.*

## Configuration file extract

---

The following listing is a sample valid configuration for a set of applications, `Server1`, `Server2`, and clients, using the leasing plug-in. This configuration is included in generated Orbix domains, `OrbixInstallDir/etc/domains/domain_name.cfg`, where `domain_name` is the name of your domain.

### Example7 Configuration File Extract for Leasing Plug-In

```
# Orbix Configuration File
...
demos {
...
  session_management
  {
    plugins:lease:shlib_name = "it_lease";
    plugins:lease:ClassName =
"com.iona.corba.plugin.lease.LeasePlugIn";
    orb_plugins = ["local_log_stream", "lease",
"iiop_profile", "giop", "iiop"];
    binding:client_binding_list = ["POA_Coloc",
"LEASE+GIOP+IIOP",
"GIOP+IIOP"];
    binding:server_binding_list = ["LEASE", ""];
    plugins:lease:allow_advertisement_overwrites = "true";
    # default is false
    event_log:filters = ["IT_LEASE=*"];
```

```
server1 {  
# client must ping every 10 seconds  
plugins:lease:lease_ping_time = "10000";  
# leases will expire after 20 seconds of inactivity  
plugins:lease:lease_reap_time = "20000";  
plugins:lease:lease_name_to_advertise  
= "PersonFactorySrv1";  
};
```

```
server2 {  
# client must ping every 20 seconds  
plugins:lease:lease_ping_time = "20000";  
# leases will expire after 40 seconds of inactivity  
plugins:lease:lease_reap_time = "40000";  
plugins:lease:lease_name_to_advertise  
= "PersonFactorySrv2";  
};  
};  
...  
};
```

# Leasing IDL Interfaces

---

*The complete IDL for the leasing plug-in.*

## The IT\_Leasing IDL module

---

The IT\_Leasing module is defined as follows:

Example8 The IT\_Leasing IDL Module

```
//IDL
#ifndef __IT_LEASING_IDL_
#define __IT_LEASING_IDL_

//
// @Copyright (c) 2000 IONA Technologies PLC. All Rights
// Reserved.

#include <omg/orb.idl>
#include <omg/IOP.idl>
#pragma prefix "iona.com"
```

```
module IT_Leasing
{
// Type definitions
typedef string LeaseID;
// Possible error conditions
exception LeaseHasExpired {};
```

```
enum LeaseAdvertisementError {
NAMING_SERVICE_UNREACHABLE,
LEASE_ALREADY_ADVERTISED,
LEASE_ALREADY_BOUND_IN_NS,
UNKNOWN_ERROR
};
```

```
exception CouldNotAdvertiseLease
{
  LeaseAdvertisementError reason;
};
```

```
exception CouldNotAcquireLease {};
```

```
// This is the maximum amount of time that a client leasing
// plugin will wait before automatically renewing a
// particular lease. The value is set in the server plugins'
// configuration.
typedef unsigned long IdleTimeBeforePing; // milliseconds
```

```
// This interface must be implemented by servers that
// wish to advertise leases.
interface LeaseCallback
{
// Informs the server that a client wants a new lease.
LeaseID acquire_lease()
raises (CouldNotAcquireLease);
```

```
// Informs the server that a lease not been renewed
// (usually because the client has gone away)
void lease_expired(
in LeaseID lease_id
);
```

```
// Informs the server that a client has explicitly  
// released a lease  
void lease_released(in LeaseID lease_id);
```

```
// renew_lease() is called by leasing plugins on the  
// client side to renew leases after some idle time.  
// This is semantically equivalent to a 'keepalive'  
// or 'heartbeat' method.  
void renew_lease(in LeaseID lease_id)  
raises (LeaseHasExpired);  
};
```

```
// This is the interface that leasing plugins will  
// expose on the server side. Server programmers must  
// interact with this interface to advertise leases.  
local interface ServerLeaseAgent  
{
```

```
// advertise_lease() is called by the server  
// to start the lease advertisement. The ping time  
// and ServerID values for the lease are obtained  
// from configuration.  
void advertise_lease(in LeaseCallback lease_callback)  
raises (CouldNotAdvertiseLease);
```

```
// Helper function that generates a system defined  
// lease id, in case the server does not need to attach  
// any specific meaning to incoming leases.  
LeaseID manufacture_lease_id();
```

```
// You may call this method at any time to withdraw your  
// lease, but note that the plugin will automatically  
// withdraw your lease at ORB shutdown time, so you  
// typically never need to call this method.  
void withdraw_lease();
```

```
// Call this method if you wish the plugin to  
// detect that a particular lease has expired (usually  
// due to non-graceful client termination).  
// The typical place to call this is from your  
// implementation of LeaseCallback::acquire_lease().  
void lease_acquired(in LeaseID lease_id);
```

```
// Call this method when you wish the plugin to stop
// detecting that a particular lease has expired, usually
// because a client has terminated gracefully and
// released the lease themselves.
// The typical place to call this is from your
// implementation of LeaseCallback::lease_released().
void lease_released(in LeaseID lease_id);
};
```

```
// This interface represents the lease details that will
// be added to requests by leasing clients. The information
// will be added as a ServiceContext and be available within
// the servant implementations through the Current interface.
local interface Current :
CORBA::Current
{
exception NoContext {};
```

```
LeaseID get_lease_id()
raises (NoContext);
```

```
};
const IOP::ServiceId SERVICE_ID = 0x49545F43;
};
#endif /*__IT_LEASING_IDL__*/
```

# Glossary

---

## A

### activator

A server host facility that is used to activate server processes.

## ART

Adaptive Runtime Technology. Rocket Software's modular, distributed object architecture, which supports dynamic deployment and configuration of services and application code. ART provides the foundation for Orbix software products.

## C

### CFR

See [configuration repository](#).

### client

An application (process) that typically runs on a desktop and requests services from other applications that often run on different machines (known as server processes). In CORBA, a client is a program that requests services from CORBA objects.

### configuration

A specific arrangement of system elements and settings.

### configuration domain

Contains all the configuration information that Orbix ORBs, services and applications use. Defines a set of common configuration settings that specify available services and control ORB behavior. This information consists of configuration variables and their values. Configuration domain data can be implemented and maintained in a centralized Orbix configuration repository or as a set of files distributed among domain hosts. Configuration domains let you organize ORBs into manageable groups, thereby bringing scalability and ease of use to the largest environments. See also [configuration file](#) and [configuration repository](#).

### configuration file

A file that contains configuration information for Orbix components within a specific configuration domain. See also [configuration domain](#).

### configuration repository

A centralized store of configuration information for all Orbix components within a specific configuration domain. See also [configuration domain](#).

### configuration scope

Orbix configuration is divided into scopes. These are typically organized into a root scope and a hierarchy of nested scopes, the fully-qualified names of which map directly to ORB names. By organizing configuration properties into various scopes, different settings can be provided for individual ORBs, or common settings for groups of ORB. Orbix services, such as the naming service, have their own configuration scopes.

## CORBA

Common Object Request Broker Architecture. An open standard that enables objects to communicate with one another regardless of what programming language they are written in, or what operating system they run on. The CORBA specification is produced and maintained by the OMG. See also [OMG](#).

## CORBA naming service

An implementation of the OMG Naming Service Specification. Describes how applications can map object references to names. Servers can register object references by name with a naming service repository, and can advertise those names to clients. Clients, in turn, can resolve the desired objects in the naming service by supplying the appropriate name. The Orbix naming service is an example.

## CORBA objects

Self-contained software entities that consist of both data and the procedures to manipulate that data. Can be implemented in any programming language that CORBA supports, such as C++ and Java.

## D

### deployment

The process of distributing a configuration or system element into an environment.

## I

### IDL

Interface Definition Language. The CORBA standard declarative language that allows a programmer to define interfaces to CORBA objects. An IDL file defines the public API that CORBA objects expose in a server application. Clients use these interfaces to access server objects across a network. IDL interfaces are independent of operating systems and programming languages.

### IIOP

Internet Inter-ORB Protocol. The CORBA standard messaging protocol, defined by the OMG, for communications between ORBs and distributed applications. IIOP is defined as a protocol layer above the transport layer, TCP/IP.

### implementation repository

A database of available servers, it dynamically maps persistent objects to their server's actual address. Keeps track of the servers available in a system and the hosts they run on. Also provides a central forwarding point for client requests. See also [location domain](#) and [locator daemon](#).

### interceptor

An implementation of an interface that the ORB uses to process requests. Abstract request handlers that can implement transport protocols (such as IIOP), or manipulate requests on behalf of a service (for example, adding transaction identity).

### Interface Definition Language

See [IDL](#).

### invocation

A request issued on an already active software component.

## IOR

Interoperable Object Reference. See [object reference](#).

## L

### location domain

A collection of servers under the control of a single locator daemon. Can span any number of hosts across a network, and can be dynamically extended with new hosts. See also [locator daemon](#) and [node daemon](#).

### locator daemon

A server host facility that manages an implementation repository and acts as a control center for a location domain. Orbix clients use the locator daemon, often in conjunction with a naming service, to locate the objects they seek. Together with the implementation repository, it also stores server process data for activating servers and objects. When a client invokes on an object, the client ORB sends this invocation to the locator daemon, and the locator daemon searches the implementation repository for the address of the server object. In addition, enables servers to be moved from one host to another without disrupting client request processing. Redirects requests to the new location and transparently reconnects clients to the new server instance. See also [location domain](#), [node daemon](#), and [implementation repository](#).

## N

### naming service

See [CORBA naming service](#).

### node daemon

Starts, monitors, and manages servers on a host machine. Every machine that runs a server must run a node daemon.

## O

### object reference

Uniquely identifies a local or remote object instance. Can be stored in a CORBA naming service, in a file or in a URL. The contact details that a client application uses to communicate with a CORBA object. Also known as interoperable object reference (IOR) or proxy.

## OMG

Object Management Group. An open membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications, including CORBA. See [www.omg.com](http://www.omg.com).

## ORB

Object Request Broker. Manages the interaction between clients and servers, using the Internet Inter-ORB Protocol (IIOP). Enables clients to make requests and receive replies from servers in a distributed computer environment. Key component in CORBA.

## P

**POA**

Portable Object Adapter. Maps object references to their concrete implementations in a server. Creates and manages object references to all objects used by an application, manages object state, and provides the infrastructure to support persistent objects and the portability of object implementations between different ORB products. Can be transient or persistent.

**S****server**

A program that provides services to clients. CORBA servers act as containers for CORBA objects, allowing clients to access those objects using IDL interfaces.

# Notices

---

## Copyright

---

© 1996-2025 Rocket Software, Inc. or its affiliates. All Rights Reserved.

## Trademarks

---

Rocket is a registered trademark of Rocket Software, Inc. For a list of Rocket registered trademarks go to: [www.rocketsoftware.com/about/legal](http://www.rocketsoftware.com/about/legal). All other products or services mentioned in this document may be covered by the trademarks, service marks, or product names of their respective owners.

## Examples

---

This information might contain examples of data and reports. The examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

## License agreement

---

This software and the associated documentation are proprietary and confidential to Rocket Software, Inc. or its affiliates, are furnished under license, and may be used and copied only in accordance with the terms of such license.

Note: This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when exporting this product.

## Corporate information

---

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: [www.rocketsoftware.com](http://www.rocketsoftware.com)

## Contacting Technical Support

---

The Rocket Community is the primary method of obtaining support. If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Community and report a problem, download an update, or read answers to FAQs. To log in to the Rocket Community or to request a Rocket Community account, go to [www.rocketsoftware.com/support](http://www.rocketsoftware.com/support). In addition to using the Rocket Community to obtain support, you can use one of the telephone numbers that are listed above or send an email to [support@rocketsoftware.com](mailto:support@rocketsoftware.com).

Rocket Global Headquarters  
77 4th Avenue, Suite 100  
Waltham, MA 02451-1468  
USA

## Country and Toll-free telephone number

---

To contact Rocket Software by telephone for any reason, including obtaining pre-sales information and technical support, use one of the following telephone numbers.

- United States: 1-855-577-4323
- Australia: 1-800-823-405
- Belgium: 0800-266-65
- Canada: 1-855-577-4323
- China: 400-120-9242
- France: 08-05-08-05-62
- Germany: 0800-180-0882
- Italy: 800-878-295
- Japan: 0800-170-5464
- Netherlands: 0-800-022-2961
- New Zealand: 0800-003210
- South Africa: 0-800-980-818
- United Kingdom: 0800-520-0439