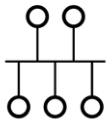

Orbix®

CORBA Session Management Guide, C++

Version 6.2, December 2004

Making Software Work Together™

Updated: 10-Apr-2007

M 3 2 1 8

Contents

Preface

This book describes the Orbix session management capability, which is based on the Orbix leasing plug-in.

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 C++ language.

Additional resources

The IONA knowledge base (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles, written by IONA experts, about the Orbix and other products. You can access the knowledge base at the following location:

The IONA update center (http://www.iona.com/support/updates/index.xml) contains the latest releases and patches for IONA products:

If you need help with this or any other IONA products, contact IONA at support@iona.com. Comments on IONA documentation can be sent to docs-support@iona.com .

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>
```

Italic

Italic words in normal text represent *emphasis* and *new terms*.

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:

```
% cd /users/your_name
```

Note: Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

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.

CHAPTER 1

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.

In this chapter

The following topics are discussed in this chapter:

The Leasing Framework

Overview

The leasing plug-in is an add-on feature for Orbix that manages server-side and client-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 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 `LeaseCallback` object to be bound in the naming service.

Stage	Description
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.
4	Upon client shut down: • If the client shuts down gracefully, the plug-in automatically releases the lease with the server. • 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.

Lease acquisition

A client initiates a session by acquiring a lease from a leasing server, as shown in Figure 1.

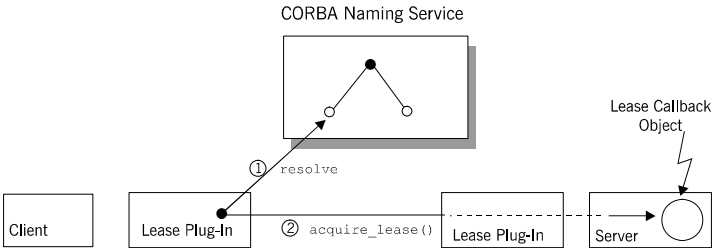

Figure 1: *The Client Acquires a Lease*

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

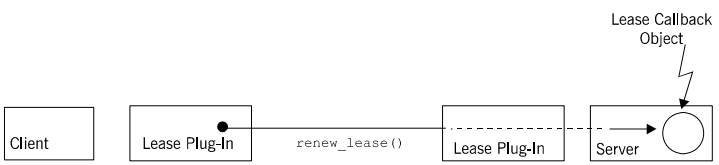

Figure 2: *The Client Renews the Lease*

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

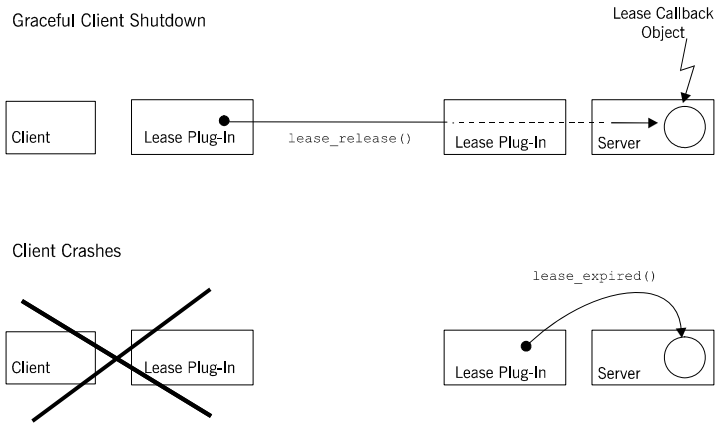

Figure 3: *The Lease is Released When the Client Shuts Down*

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/6.2/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 `Person` objects by calling the `create_person()` operation, with unique `name` arguments for each `Person`.

Stage	Description
2	When a client terminates, the `Person` objects it created no longer need to be held inside the server memory and are deleted.

Using the Leasing Plug-In on the Server Side

Overview

This section explains how to configure and program a server to use the session management features of the leasing plug-in.

In this section

This section contains the following subsections:

Overview of Server-Side Leasing

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" on page 49.

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()`.

ServerLeaseAgent 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()`.

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 LeaseCallback Interface

Overview

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 1: *Object Instances Used in the LeaseCallbackImpl Class*

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

Implementation code

The `IT_Leasing::LeaseCallback` interface is implemented by the `LeaseCallbackImpl` C++ class, as shown in Example 1.

Example 1: *The LeaseCallbackImpl Class*

```
//C++
char*
1 LeaseCallbackImpl::acquire_lease()
    IT_THROW_DECL((CORBA::SystemException,
                   IT_Leasing::CouldNotAcquireLease))
{
    CORBA::String_var new_lease =
     leaseObj->manufacture_lease_id();
    // inform the plugin that it should monitor the lifecycle
    // and status of this new lease
    leaseObj->lease_acquired(new_lease);
    return new_lease._retn();
}
2 void LeaseCallbackImpl::lease_expired(const char* lease_id)
   IT_THROW_DECL((CORBA::SystemException))
{
    m_factory->owner_has_gone_away(lease_id);
}
3 void LeaseCallbackImpl::lease_released(const char* lease_id)
   IT_THROW_DECL((CORBA::SystemException))
{
    leaseObj->lease_released(lease_id);
    m_factory->owner_has_gone_away(lease_id);
}
4 void LeaseCallbackImpl::renew_lease(const char* lease_id)
   IT_THROW_DECL((CORBA::SystemException,
                  IT_Leasing::LeaseHasExpired))
{
    // Nothing to do, since the plugin has already intercepted
    // this request and knows that the lease has been renewed.
}
```

The code can be explained as follows:

1. The `LeaseCallbackImpl::acquire_lease()` function 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()` function 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" on page 41). 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` deletes the `Person` objects completely at this point. Alternatively, a server could *evict* the transient objects by persisting their data before physically deleting them from memory.

3. The `LeaseCallbackImpl::lease_released()` function 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()` function 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.

Tracking Sessions in the Server

Overview

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` C++ class as shown in Example 2.

Example 2: *The PersonFactoryImpl Class (Sheet 1 of 3)*

```
// C++
LeaseTest::Person_ptr
  PersonFactoryImpl::create_person(const char* name)
 IT_THROW_DECL((CORBA::SystemException,
                LeaseTest::PersonAlreadyExists))
{
    LeaseTest::Person_var result = LeaseTest::Person::_nil();
    try
    {
1       CORBA::String_var owner = CORBA::string_dup("<unknown>");
        try
        {
2           CORBA::Object_var objref =
                global_orb->resolve_initial_references(
                                "LeaseCurrent"
                            );
            if (!CORBA::is_nil(objref))
            {
                IT_Leasing::Current_var current =
                    IT_Leasing::Current::_narrow(objref);
                if (!CORBA::is_nil(current))
                {
```

Example 2: *The PersonFactoryImpl Class (Sheet 2 of 3)*

```
3                       owner = current->get_lease_id();
                    }
                }
            }
            catch (IT_Leasing::Current::NoContext &)
            {
                cerr << "Couldn't find the relevant "
                     << "ServiceContext data." << endl;
            }
            catch (...)
            {
                cerr << "An unknown exception occurred while "
                     << "getting ServiceContext data." << endl;
            }

            // Create a new Person servant and activate it
            PersonImpl*                  newPersonServant;
            PortableServer::ObjectId_var  oid;
            CORBA::Object_var             tmp_ref;
            ...
            // Assume that we have already checked that the
            // person does not exist, so it is created and
            // stored with the others, indexed by its name
            //
4           newPersonServant = new PersonImpl(name, owner);
            oid = m_poa->activate_object(newPersonServant);
            tmp_ref = m_poa->id_to_reference(oid);
            result = LeaseTest::Person::_narrow(tmp_ref);
            assert(!CORBA::is_nil(result));
            // Store the new servant with the others
            IT_String temp_string(name);
5           m_People[temp_string] = newPersonServant;
            dump_people_to_screen();
        }
        catch (const CORBA::SystemException &se)
        {
            cerr << se << endl;
        }
        catch (...)
        {
            cerr << "Unknown exception within create_person()"
                 << endl;
        }
6       return result._retn();
}
```

Example 2: *The PersonFactoryImpl Class (Sheet 3 of 3)*

```
7   void PersonFactoryImpl::owner_has_gone_away(const char* owner)
    {
        // Iterate through the people map and evict any people
        // who were created by 'owner'.
        //
        IT_Locker <IT_Mutex> lock(m_mutex);
        IT_String current_name;
        People::iterator theIter = m_People.begin();
        while (theIter != m_People.end())
        {
            current_name = (*theIter).second->owner();
8           if (current_name == owner)
            {
                // deactivate the servant before deleting it
                 PortableServer::ObjectId_var oid =
                    m_poa->servant_to_id((*theIter).second);
                // deactivate the servant with the corresponding
                // id on the POA
9               m_poa->deactivate_object(oid);
                cout << "Deleting: " << (*theIter).first << endl;
10              delete (*theIter).second;
                m_People.erase(theIter);
                theIter = m_People.begin(); //iterator is invalidated
                continue;
            }
            theIter++;
        }
        dump_people_to_screen();
    }
    ...
```

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()` function 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()` function 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. String comparison between `current_name` and `owner` can be performed using `==` because `current_name` is declared to be of `IT_String` type, which has similar properties to the `std::string` type from the C++ standard template library.
9. Before deleting a `Person` object, the corresponding servant must be deactivated by calling `PortableServer::POA::deactivate_object()`.
10. The servant object and its corresponding `m_People` entry are deleted in this and the following lines of code.

Advertising 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()` function. 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()` function to advertise the lease:

Example 3: *Advertising the Lease in the main() Function (Sheet 1 of 2)*

```
// C++
...
int
main(int argc, char **argv)
{
    // Assume that we have already created and activated a
    // LeaseCallback servant and created a reference for it
    // called theLeaseCallbackObj.
    ...
    // Contact the lease plugin and advertise a lease
    try
    {
1       CORBA::Object_var tmp_ref =
  global_orb->resolve_initial_references("IT_ServerLeaseAgent");
        leaseObj =
   IT_Leasing::ServerLeaseAgent::_narrow(tmp_ref);
```

Example 3: *Advertising the Lease in the main() Function (Sheet 2 of 2)*

```
2           leaseObj->advertise_lease(theLeaseCallbackObj);
        }
        catch (IT_Leasing::CouldNotAdvertiseLease &ex)
        {
          // process the exception
        }
        catch (CORBA::Exception &e)
        {
          // ...
        }
        ...
};
```

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, `theLeaseCallbackObj`, which causes the `LeaseCallback` object to be registered with the plug-in.

Configuring the Server

Overview

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 2: *Configuration Variables Used on the Client Side*

Configuration Variable	Purpose
`binding:server_binding_list`	The server binding list is modified, instructing the ORB to insert `LEASE` interceptors into server-side bindings.
`plugins:lease:` `lease_name_to_advertise`	The name under which the `LeaseCallback` object is bound in the naming service. This name must be unique per server.
`plugins:lease:lease_ping_time`	The time interval (in milliseconds) between successive ping messages sent by client-side plug-ins to renew the lease.
`plugins:lease:lease_reap_time`	If a particular client's lease is not pinged within `lease_reap_time`, 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" on page 41.

Example configuration

For a complete example of a client-side and server-side configuration, see "Sample Leasing Plug-In Configuration" on page 45.

Using the Leasing Plug-In on the Client Side

Overview

This section explains how to configure and program a server to use the session management features of the leasing plug-in.

In this section

This section contains the following subsections:

Overview of Client-Side Leasing

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

There are two approaches to using the leasing plug-in on the client side, as follows:

- *Configuration only*—no modifications to the client code are required. This approach enables you to manage session resources on the server side of an application. Whenever a client session ends, the server can automatically clean up associated session resources. See "Configuring the Client" on page 25 for details.
- *Configuration and programming*—if you need to manage session resources on the client side as well, it is necessary to modify the client code, as described in "Tracking Sessions in the Client" on page 26, "Implementing the ClientLeaseCallback Interface" on page 30, and "Activating and Registering the Client Callback" on page 33.

IT_Leasing module for the client

Example 4 shows an extract from the `IT_Leasing` module, showing the interfaces that are relevant to programming on the client side of session management application.

Example 4: *IT_Leasing Module for the Client*

```
// IDL
IT_Leasing
{
    interface ClientLeaseCallback
    {
        void lease_started(
            in string lease_id,
            in string server_lease_id
        );

        void lease_renewal_failed(
            in string lease_id,
```

Example 4: *IT_Leasing Module for the Client*

```
            in string server_lease_id
        );

        void lease_stopped(
            in string lease_id,
            in string server_lease_id
        );
    };

    local interface ClientLeaseAgent
    {
        void register_lease_callback(
            in ClientLeaseCallback client_lease_callback
        ) raises (CouldNotRegisterLeaseCallback);
    };

    local interface Current :
    CORBA::Current
    {
        exception NoContext {};

        LeaseID get_lease_id() raises (NoContext);
    };

    local interface Current2 :
    IT_Leasing::Current
    {
        ServerID get_server_id() raises (NoContext);
    };
};
```

ClientLeaseCallback interface

The client must provide an implementation of the `IT_Leasing::ClientLeaseCallback` interface to receive notifications of lease-related events from the leasing plug-in. For example, if a connection to a server is lost, the plug-in calls back on `IT_Leasing::ClientLeaseCallback::lease_stopped()`.

ClientLeaseAgent interface

The implementation of the `IT_Leasing::ClientLeaseAgent` interface is provided by the leasing plug-in. The client uses this interface to register a cleint lease callback object with the plug-in.

Current2 interface

The client accesses the `IT_Leasing::Current2` interface to obtain the lease ID (by calling `get_lease_id()`) and the server ID (by calling `get_server_id()`) associated with the current session. The returned lease ID and server ID refer to the session associated with the most recently invoked-upon proxy object.

Configuring the Client

Configuration variables

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

Table 3: *Configuration Variables Used on the Client Side*

Configuration Variable	Purpose
`plugins:lease:shlib_name`	Identifies the shared library that contains the plug-in code.
`orb_plugins`	The ORB plug-in list is modified to ensure that the lease plug-in is automatically loaded when the client ORB is initialized.
`binding:client_binding_list`	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" on page 41.

Configuring for colocated 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 colocated 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";
orb_plugins = ["local_log_stream", "lease", "iiop_profile",
   "giop", "iiop"];
binding:client_binding_list = ["POA_Coloc", "LEASE+GIOP+IIOP",
   "GIOP+IIOP"];
```

Tracking Sessions in the Client

Overview

In order to manage session resources on the client side, the first prerequisite is to have some way of identifying the current session. You can then associate any session resources with the relevant session identifiers (for example, storing resources in a hash map, where the session identifier is used as the key).

This section explains how to use the leasing programming interface to identify the current session on the client side.

Identifying sessions on the client side

In order to identify a session uniquely on the client side, you need both the current lease ID and the current server ID. The IDs have the following significance on the client side:

- *Server ID*—uniquely identifies a server with which the client has a connection.
- *Lease ID*—used in combination with the server ID to identify a session uniquely. Servers allocate a distinct lease ID for each established connection.

Because a client can open multiple connections to a single server, the server ID alone is *not* sufficient to identify a session uniquely. In scenarios where the client opens multiple connections to the server, the lease ID is used to distinguish between the different connctions.

You can obtain the server ID and lease ID for a particular connection by accessing the `IT_Leasing::Current2` interface immediately after invoking an operation on a proxy object associated with that connection.

IT_Leasing::Current2 interface

Example 5 shows the `Current` interfaces from the `IT_Leasing` module. The `IT_Leasing::Current2` (which inherits from `IT_Leasing::Current`) provides both the `get_server_id()` operation and the `get_lease_id()` operation.

Example 5: *The IT_Leasing Current Interfaces*

```
// IDL
module IT_Leasing {
    local interface Current :
```

Example 5: *The IT_Leasing Current Interfaces*

```
    CORBA::Current
    {
        exception NoContext {};

        LeaseID
        get_lease_id() raises (NoContext);
    };

    local interface Current2 :
    IT_Leasing::Current
    {
        ServerID
        get_server_id() raises (NoContext);
    };
};
```

Tracking sessions using the current lease ID and server ID

Example 6 shows an example of how to track session resources on the client side using the leasing plug-in (based on the leasing demonstration).

Example 6: *Tracking Session Resources in the Client*

```
// C++
LeaseTest::Person_var newPerson1;
CORBA::String_var lease_id;
CORBA::String_var server_id;

newPerson1 = factory1->create_person("Fred_Flintstone");

// Get IDs for the current connection
lease_id = get_lease_id();
server_id = get_server_id();

// Cache the newPerson1 object
add_session_resource(newPerson1, server_id, lease_id);
```

The `factory1` object is a proxy for the `LeaseTest::PersonFactory` IDL interface. Immediately after invoking the `create_person()` operation on the `factory1` object, the server ID and lease ID for this connection can be retrieved from the `IT_Leasing::Current2` object (see "Obtaining the lease ID" on page 28 and "Obtaining the server ID" on page 29 for the implementation of the `get_lease_id()` and `get_server_id()` functions).

Once you have the server ID and lease ID, you can track resources for this session. For example, if you decided to cache a copy of the `Person` object, `newPerson1`, you might define a function, `add_session_resource()`, that associates the cached data with the current server ID and lease ID.

Obtaining the lease ID

Example 7 shows you how to obtain the current lease ID by querying the `IT_Leasing::Current2` object.

Example 7: *Extracting the Lease ID from IT_Leasing::Current2*

```
// C++
char*
get_lease_id()
{
    try
    {
        CORBA::Object_var objref =

   global_orb->resolve_initial_references("LeaseCurrent");
        if (!CORBA::is_nil(objref))
        {
            IT_Leasing::Current2_var current =
                IT_Leasing::Current2::_narrow(objref);
            if (CORBA::is_nil(current))
            {
                return 0;
            }

            return current->get_lease_id();
        }
    }
    catch (IT_Leasing::Current::NoContext &)
    {
        cerr << "Couldn't find the relevant ServiceContext data."
            << endl;
    }
    catch (...)
    {
        cerr << "An unknown exception occurred while getting
   ServiceContext data." << endl;
    }
    return 0;
}
```

Obtaining the server ID

Example 8 shows you how to obtain the current server ID by querying the `IT_Leasing::Current2` object.

Example 8: *Extracting the Server ID from IT_Leasing::Current2*

```
// C++
char*
get_server_id()
{
    try
    {
        CORBA::Object_var objref =

   global_orb->resolve_initial_references("LeaseCurrent");
        if (!CORBA::is_nil(objref))
        {
            IT_Leasing::Current2_var current =
                IT_Leasing::Current2::_narrow(objref);
            if (CORBA::is_nil(current))
            {
                return 0;
            }

            return current->get_server_id();
        }
    }
    catch (IT_Leasing::Current::NoContext &)
    {
       cerr << "Couldn't find the relevant ServiceContext data."
   << endl;
    }
    catch (...)
    {
        cerr << "An unknown exception occurred while getting
   ServiceContext data." << endl;
    }
    return 0;
}
```

Implementing the ClientLeaseCallback Interface

Overview

You can optionally implement the `ClientLeaseCallback` interface in a leasing client, if you are interested in receiving notifications about session lifecycles. In particular, you can use the client lease callback to manage session-related resources on the client side.

ClientLeaseCallback implementation class

Example 9 shows the `ClientLeaseCallbackImpl` class, which implements the `ClientLeaseCallback` IDL interface (see Example 4 on page 22).

Example 9: *The ClientLeaseCallbackImpl Implementation*

```
// C++
#include <omg/orb.hh>
#include "client_lease_callback_impl.h"

IT_USING_NAMESPACE_STD

// ClientLeaseCallbackImpl constructor
ClientLeaseCallbackImpl::ClientLeaseCallbackImpl(
)
{}

// ClientLeaseCallbackImpl destructor.
ClientLeaseCallbackImpl::~ClientLeaseCallbackImpl()
{}

// lease_started() -- Implements IDL operation
// "IT_Leasing::CLeaseCallback::lease_started".
//
void
1 ClientLeaseCallbackImpl::lease_started(
    const char* lease_id,
    const char* server_id
)
    IT_THROW_DECL((CORBA::SystemException))
{
    cout << "\tA lease has started with the following details:"
   << endl;
    cout << "\tServer ID: " << server_id << ", Lease ID: " <<
   lease_id << endl;
}
```

Example 9: *The ClientLeaseCallbackImpl Implementation*

```
// register_lease_callback() -- Implements IDL operation
// "IT_Leasing::CLeaseCallback::lease_renewal_failed".
//
void
2 ClientLeaseCallbackImpl::lease_renewal_failed(
    const char* lease_id,
    const char* server_id
)
    IT_THROW_DECL((CORBA::SystemException))
{
    cout << "\tA lease with the following details has failed to
   renew: " << endl;
    cout << "\tServer ID: " << server_id << ", Lease ID: " <<
   lease_id << endl;
}

void
3 ClientLeaseCallbackImpl::lease_stopped(
    const char* lease_id,
    const char* server_id
)
    IT_THROW_DECL((CORBA::SystemException))
{
    cout << "\tThe lease has been released with the following
   details:" << endl;
    cout << "\tServer ID: " << server_id << ", Lease ID: " <<
   lease_id << endl;
}
```

The preceding implementation code can be explained as follows:

1. The lease plug-in calls `lease_started()` when a new lease has been acquired from a leasing server, indicating that a new session has started. The new session is uniquely identified by the combination of a lease ID, `lease_id`, and a server ID, `server_id`.
2. The lease plug-in calls `lease_renewal_failed()`, if the remote server refuses to renew the client's lease. For example, when the client's lease plug-in calls the server's heartbeat operation, `IT_Leasing::LeasCallback::renew_lease()`, the server might throw the `IT_Leasing::LeaseHasExpired` exception instead of renewing the lease.

Upon receiving this callback notification, the client should clean up any resources associated with the session identified by `lease_id` and `server_id`.

3. The lease plug-in calls `lease_stopped()`, if a session becomes unavailable for any reason other than a failed renewal—for example, if the server closes the connection or if the server shuts down.

   Upon receiving this callback notification, the client should clean up any resources associated with the session identified by `lease_id` and `server_id`.

Activating and Registering the Client Callback

Overview

In order to start receiving notifications from the leasing plug-in, it is necessary both to *activate* and *register* the client lease callback object. These steps can be described as follows:

- *Activation*—is the same set of programming steps that you usually use on the server side to activate a CORBA object. Although the client callback object is only used locally, you still have to perform the same activation steps that you would use for a fully-fledged CORBA object.
- *Registration*—before the callback can receive notifications from the leasing plug-in, it is necessary for the plug-in to be aware of the existence of the callback object. Therefore, you must register the callback object with the leasing plug-in by obtaining a reference to an `IT_Leasing::ClientLeaseAgent` instance and then calling the `register_lease_callback()` operation.

ClientLeaseAgent interface

Example 10 shows the IDL for the `IT_Leasing::ClientLeaseAgent` interface. This interface exposes a single operation, `register_lease_callback()`, that is used to register a client lease callback object.

Example 10: *The IT_Leasing::ClientLeaseAgent Interface*

```
// IDL
...
module IT_Leasing {
    local interface ClientLeaseAgent
    {
        void
        register_lease_callback(
            in ClientLeaseCallback client_lease_callback
        ) raises (CouldNotRegisterLeaseCallback);
    };
};
```

ClientLeaseAgent initial reference string

In order to obtain a `ClientLeaseAgent` instance, you invoke the `CORBA::ORB::resolve_initial_references()` operation, passing in the `IT_ClientLeaseAgent` initial reference string. For example:

```
// C++
CORBA::Object_var obj;
obj = orb->resolve_initial_references("IT_ClientLeaseAgent");
```

Activating and registering the client callback object

Example 11 shows the code from the client `main()` function that activates and registers a client callback object. Once the callback object is activated and registered, it is then ready to receive notifications from the lease plug-in.

Example 11: *Activating and Registering a Client Leasing Callback*

```
// C++
PortableServer::ObjectId_var oid; // For client side POA

CORBA::Object_var tmp_ref;  // For temporary object references.

// Initialise the ORB
global_orb = CORBA::ORB_init(argc, argv,
   "demos.session_management");

// Register a callback Object with the Leasing Plugin
// to notify us when a server resource is no longer
// contactable.

1  tmp_ref = global_orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa =
   PortableServer::POA::_narrow(tmp_ref);
assert(!CORBA::is_nil(root_poa));

2  PortableServer::POAManager_var root_poa_manager =
   root_poa->the_POAManager();
assert(!CORBA::is_nil(root_poa_manager));

IT_Leasing::ClientLeaseAgent_var leaseClientAgentObj = 0;
IT_Leasing::ClientLeaseCallback_var
   the_ClientLeaseCallbackObject = 0;

3  the_ClientLeaseCallbackServant = new ClientLeaseCallbackImpl();

4  oid = root_poa->activate_object(the_ClientLeaseCallbackServant);
tmp_ref = root_poa->id_to_reference(oid);
```

Example 11: *Activating and Registering a Client Leasing Callback*

```
the_ClientLeaseCallbackObject =
   IT_Leasing::ClientLeaseCallback::_narrow(tmp_ref);

5  tmp_ref = global_orb->resolve_initial_references(
              "IT_ClientLeaseAgent"
          );
leaseClientAgentObj =
   IT_Leasing::ClientLeaseAgent::_narrow(tmp_ref);
if (CORBA::is_nil(leaseClientAgentObj))
{
    cerr << "Exiting..." << endl;
    return 1;
}

// Register the callback with the leasing plug-in
6  leaseClientAgentObj->register_lease_callback(
    the_ClientLeaseCallbackObject
);

7  root_poa_manager->activate();
...
```

The preceding code example can be explained as follows:

1. Obtain a reference to the root POA. In this example, the client lease callback object is activated by the root POA. It so happens that the root POA's default policies are appropriate for activating a callback object.
2. The root POA manager is needed later in order to complete activation of the root POA.
3. Create an instance of the client lease callback servant object, `the_ClientLeaseCallbackServant`.
4. Activate the client lease callback object on the root POA. Because the root POA's ID assignment policy is `SYSTEM_ID`, it will automatically generate an object ID, `oid`, for the callback object. From this object ID, you can then generate an object reference, `the_ClientLeaseCallbackObject`.
5. Obtain a reference to the `IT_Leasing::ClientLeaseAgent` object by resolving the initial reference string, `IT_ClientLeaseAgent`.
6. Register the callback object with the leasing plug-in by calling `register_lease_callback()` on the client lease agent object.

7. Complete the activation of the POA by calling `activate()` on the root POA manager object.

Activating the callback object in a mid-tier server

A special case arises when you want to register a client lease callback in a program that is simultaneously acting as a leasing client *and* a leasing server. For example, this case can arise in a mid-tier server, when the application is set up as follows:

- *First tier (client)*—is configured as a leasing client. In particular, the `binding:client_binding_list` variable is configured to load the `LEASE` interceptor.
- *Second tier (mid-tier server)*—is configured both as a leasing client and as a leasing server. In particular, both the `binding:client_binding_list` variable *and* the `binding:server_binding_list` variable are configured to load the `LEASE` interceptor.
- *Third tier (target server)*—is configured as a leasing server. In particular, the `binding:server_binding_list` variable is configured to load the `LEASE` interceptor.

Now if you try to register a client lease callback in the mid-tier server a potential problem arises. Because the mid-tier server is configured as a leasing server, the leasing plug-in automatically attempts to modify the callback's object reference by inserting a leasing IOR profile. To avoid this, you should activate the callback object with a POA that has been configured to suppress these IOR modifications—see "Disabling Session Management Selectively" on page 37.

Disabling Session Management Selectively

Overview

Normally, session management is enabled for *all* CORBA objects in a server as long as the `LEASE` interceptor is included in the server binding list, `binding:server_binding_list`. Conversely, session management would be disabled for all CORBA objects in a server, if the `LEASE` interceptor is omitted from the server binding list.

Sometimes, however, you might require some CORBA objects in a server to use session management, whilst others have session management disabled. To accomodate this scenario, it is possible to disable session management selectively by applying the `LeasingRequiredPolicy` to a POA instance. The `LeasingRequiredPolicy` can be set to one of the following boolean values:

- *True— (default value)* enable session management. POAs governed by this policy generate IORs that contain an additional leasing IOR component.
- *False*—disable session management. POAs governed by this policy do *not* add leasing IOR components to the IOR.

If you create a POA that has the `LeasingRequiredPolicy` policy set to false, any CORBA objects activated by that POA will have session management disabled.

The LeasingRequiredPolicy

The `IT_Leasing::LeasingRequiredPolicy` is defined by the following IDL fragment from the `IT_Leasing` module:

Example 12: *The IT_Leasing::LeasingRequiredPolicy Policy*

```
// IDL
...
module IT_Leasing
{
    const CORBA::PolicyType LEASING_POLICY_ID = 0x49545F6A;

    local interface LeasingRequiredPolicy : CORBA::Policy
    {
       // A value of True enables leasing IOR changes, a value of
       // False will disable them.
       readonly attribute boolean should_lease;
```

Example 12: *The IT_Leasing::LeasingRequiredPolicy Policy*

```
    };
};
```

To create an instance of a `LeasingRequiredPolicy` policy, call the `CORBA::ORB::create_policy()` operation, passing `IT_Leasing::LEASING_POLICY_ID` as the first argument and an `any` containing either a true or a false boolean value as the second argument.

Creating a POA with the LeasingRequiredPolicy

Example 13 shows some sample code that you can use to create a *non-leasing POA*—that is, a POA whose CORBA objects do *not* use the session management feature. Session management is disabled by setting the `LeasingRequiredPolicy` policy to `false` in the POA.

Example 13: *Creating a POA that Disables Leasing*

```
// C++
PortableServer::POA_ptr
create_non_leasing_poa(
    const char*                  poa_name,
    PortableServer::POA_ptr      parent_poa,
    PortableServer::POAManager_ptr poa_manager
)
{
    CORBA::PolicyList policies;
    policies.length(2);
    int i = 0;

    CORBA::Any_var any;
    CORBA::Boolean policy_val = IT_FALSE;
    any <<= CORBA::Any::from_boolean(policy_val);
    policies[i++] = global_orb->create_policy(
        IT_Leasing::LEASING_POLICY_ID,
        any
    );

    policies[i++] = parent_poa->create_thread_policy(
        PortableServer::ORB_CTRL_MODEL
    );
    assert(i==2);

    return parent_poa->create_POA(
        poa_name,
```

Example 13: *Creating a POA that Disables Leasing*

```
        poa_manager,
        policies
    );
}
```

APPENDIX A

Leasing Plug-In Configuration Variables

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

In this appendix

This appendix contains the following sections:

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 *CORBA 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:shlib_name Identifies the base name of the leasing plug-in shared library. The `shlib_name` variable should be set to `it_lease`.

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.

Example 14: *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:

Example 15: *The IT_Leasing Module*

```
// IDL
#pragma IT_SystemSpecification
#include <omg/orb.idl>
#include <omg/IOP.idl>
#include <orbix_pdk/policy.idl>

#pragma prefix "iona.com"

module IT_Leasing
{
    // Type definitions
    //
    typedef string LeaseID;
    typedef string ServerID;

    // Possible error conditions
    //
    exception LeaseHasExpired {};

    enum LeaseAdvertisementError {
        NAMING_SERVICE_UNREACHABLE,
```

Example 15: *The IT_Leasing Module*

```
        LEASE_ALREADY_ADVERTISED,
        LEASE_ALREADY_BOUND_IN_NS,
        UNKNOWN_ERROR
    };

    exception CouldNotAdvertiseLease
    {
        LeaseAdvertisementError reason;
    };

    exception CouldNotAcquireLease {};

    exception CouldNotRegisterLeaseCallback {};

    // 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
        );
```

Example 15: *The IT_Leasing Module*

```
        // 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
```

Example 15: *The IT_Leasing Module*

```
        // 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 must be implemented to allow client
    // callbacks from the leasing plugin
    interface ClientLeaseCallback
    {
        // Call this method when a lease starts
        //
        void
        lease_started(
            in string lease_id,
            in string server_lease_id
        );

        // Call this method when a lease fails to renew
        //
        void
        lease_renewal_failed(
            in string lease_id,
```

Example 15: *The IT_Leasing Module*

```
            in string server_lease_id
        );

        void
        lease_stopped(
            in string lease_id,
            in string server_lease_id
        );

    };

    // This is the interface that the leasing plugin will expose
    // to the client side
    local interface ClientLeaseAgent
    {
        // register_lease_callback is called by the client to
        // register a lease callback object with the leasing
        // plugin.
        void
        register_lease_callback(
            in ClientLeaseCallback client_lease_callback
        ) raises (CouldNotRegisterLeaseCallback);
    };

    // The following Policy definition can be used to prevent the
    // leasing information being placed into IORs, since there
    // can be a need to export object references  that do not have
    // leasing information within them (for instance, callback
    // objects within leasing clients).

    const CORBA::PolicyType LEASING_POLICY_ID = 0x49545F6A;

    local interface LeasingRequiredPolicy : CORBA::Policy
    {
         // A value of True enables leasing IOR changes, a value
        // of False will disable them.
        readonly attribute boolean should_lease;
    };

    // 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.
    //
```

Example 15: *The IT_Leasing Module*

```
    local interface Current :
    CORBA::Current
    {
        exception NoContext {};

        LeaseID
        get_lease_id(
        ) raises (NoContext);

    };

    local interface Current2 :
    IT_Leasing::Current
    {
        ServerID
        get_server_id(
        ) raises (NoContext);

    };

    const IOP::ServiceId SERVICE_ID = 0x49545F43;
};
```

Glossary

A

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

ART
Adaptive Runtime Technology. IONA's modular, distributed object architecture, which supports dynamic deployment and configuration of services and application code. ART provides the foundation for IONA 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 centralised Orbix configuration repository or as a set of files distributed among domain hosts. Configuration domains let you organise 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 centralised 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 organising 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.

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.

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.

---

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.

Index

O

P

R

S

T