

How to use the omniORB2 LifeCycle Support

Duncan Grisby

4 December 1997

1 Introduction

omniORB2 provides support for LifeCycle operations on objects. It is important to realise that it does not, however, provide an implementation of the CORBA services CosLifeCycle service. What it *does* provide is a means of moving object implementations from location to location, and deleting those implementations.

Using the LifeCycle support is very simple. The following sections explain the example in `src/examples/lifecycle`.

2 LifeCycle Basics

The location at which an object is first created is called the object's *home* location. The home location is guaranteed to know the object's current location—it directs clients to the correct object wherever it may be. The home location does not act as a proxy—once it has told a client where an object currently resides, the client does not need to refer to the home location until the object moves again.

Moving an object from one location to another actually involves creating a new object with the same interface and the same internal state as the one to be moved, telling the home location about the new object, then deleting the original object. Since it is impossible for the ORB to know what internal state is held by objects, it is up to the implementation to create the new object. It usually does this using a *factory*.

Since it is up to the implementation to copy objects, it is possible for an object to be moved to one with a different implementation, as long as it still has the same interface.

3 The Example

The example in `examples/lifecycle` defines an interface named `EchoLag`. It declares the operation `echoStringLag()` which takes a string as its argument, and returns the string it received on its previous call. So, on its first call it returns an undefined string, on its second call it returns the string it was given on the first call, and so on.

There are three separate programs defined in the example: `lcserver` provides an `EchoLag` object and an `EchoLagFactory` object; `lcclient` calls the functions of `EchoLag` objects and can trigger location changes; `lcremove` calls an `EchoLag`'s `remove()` operation.

4 IDL Definition

The file `echolag.idl` contains:

```
interface omniLifeCycleInfo ;
interface EchoLagFactory ;

interface EchoLag {
    string echoStringLag(in string str);
    void move(in EchoLagFactory there);
    void remove();
};
interface EchoLagFactory {
    EchoLag new_EchoLag();
    EchoLag copy_EchoLag(in string str,
                        in omniLifeCycleInfo li);
};
```

There is nothing spectacular about the `EchoLag` interface. `EchoLagFactory` has two operations, one for creating brand new `EchoLags`, and the other for creating copies of existing ones. The `copy_EchoLag()` operation takes an extra argument of an `omniLifeCycleInfo` object reference; this reference is essential, but it is not necessary to know what the interface's definition is.

5 The Implementation Class

Here's what `EchoLag_i` looks like:

```
class EchoLag_i :
    public virtual _lc_sk_EchoLag,
    public virtual omniLC::_threadControl
{
private:
    char *s;

public:
    EchoLag_i(const char *initial);
    virtual ~EchoLag_i();

    virtual char *echoStringLag(const char *str);
    virtual void move(EchoLagFactory_ptr there);
    virtual void remove();
};
```

As you can see, rather than deriving from the `_sk_EchoLag` skeleton class, it derives from `_lc_sk_EchoLag`. This is where the extra LifeCycle control functions come from. Don't worry about the `omniLC::_threadControl` class for now.

6 Moving Objects

To perform a move, we need to use two new functions. Here is the code:

```
void
EchoLag_i::move(EchoLagFactory_ptr there) {
    omniLC::ThreadLC zz(this);

    // Create a copy of ourselves:
    char *p = CORBA::string_dup(s);
    EchoLag_var newEchoLag = there->copy_EchoLag(p, _get_lifecycle());

    // Tell the system to do the move:
    _move(newEchoLag);

    // Dispose of this object:
    CORBA::BOA::getBOA()->dispose(this);
}
```

You may have been wondering where we were going to get the `omniLifeCycle-Info` object reference required by `copy_EchoLag()` from. Well, you can see that there is a new function called `_get_lifecycle()` which does just that. You still don't need to worry about what it does.

Once we have made a copy of our object, we just need to tell the system about it—that is what the `_move()` function does.

The `omniLC::ThreadLC` bit will be explained later.

7 Removing Objects

Removing an object is simply a case of using the `_remove()` function:

```
void
EchoLag_i::remove() {
    omniLC::ThreadLC zz(this);
    _remove();
    CORBA::BOA::getBOA()->dispose(this);
}
```

8 Implementing the Factory

The factory is just a normal CORBA object, so there is nothing unusual about its class definition. However, since the object it is creating is a bit special, it needs to

be careful to do the right thing.

8.1 New Objects

Here's the code for `new_EchoLag()`:

```
EchoLag_ptr
EchoLagFactory_i :: new_EchoLag() {
    EchoLag_i *elag = new EchoLag_i("First call");
    elag->_obj_is_ready(elag->_boa());

    return elag->_this();
}
```

It creates the new object in the expected way (defining the ‘undefined’ string in the process). The *vitaly important* bit is that it returns the result of `_this()` rather than using `_duplicate()`. If you do this wrong, Lifecycle operations will not work.

8.2 Copied Objects

Implementing `copy_EchoLag()` is a little more complicated:

```
EchoLag_ptr
EchoLagFactory_i :: copy_EchoLag(const char *str,
                                omniLifecycleInfo_ptr li) {
    EchoLag_i *elag = new EchoLag_i(str);
    elag->_set_lifecycle(li);
    elag->_obj_is_ready(elag->_boa());
    return EchoLag::_duplicate(elag);
}
```

Here we create a new `EchoLag` with the string from the old one, then call another new function, `_set_lifecycle()`. This tells the system that the object is a moved version of another one. Finally, this time it is essential that `_duplicate()` is used for the return value rather than `_this()`. Getting it wrong will confuse the system.

8.3 What's going on?

When you create a brand new object, the system must create the things it uses to track the object from location to location. It does this on the first call the `_this()`. The object reference returned by `_this()` is not really the implementation object, but a *wrapper* object pretending to be it. It is this object which keeps track of the implementation object's current location, and lets clients know about it. Subsequent calls to `_this()` return the wrapper object.

When you create a copy of an object, the call to `_set_lifecycle()` sets the object's home location wrapper object. So, calls to `_this()` return the reference of the wrapper object—just what we want if we are exporting an object reference from the implementation, but just what we do not want when creating the object.

9 Concurrency Control

Since everything is happening in a multi-threaded environment, some concurrency control is required to prevent inconsistent states arising while objects are copied and moved. Such concurrency control is up to the implementation, but most implementations will make use of the `omniLC::_threadControl` class provided.

The easiest way to use the facilities of the `omniLC::_threadControl` class is to use the two helper classes, `omniLC::ThreadOp` and `omniLC::ThreadLC`. Normal operations create an `omniLC::ThreadOp` object on their stack; LifeCycle operations like `move()` and `remove()` create `omniLC::ThreadLC` objects on their stack (as seen above).

10 Exceptions

The LifeCycle support adds no new exceptions, but it does make use of an obscure standard one—it is common for operations to throw `CORBA::TRANSIENT` exceptions. This happens when an object moves from one non-home location to another: each client keeps track of where it thinks the object currently resides; if the client tries an operation invocation and the object does not exist at that location, it resets its location to the object's home location and throws a `TRANSIENT` exception.

By default, `omniORB` provides a `TRANSIENT` exception handler which keeps retrying the operation with an ever-increasing delay between retries. You can provide your own exception handler with `installTransientExceptionHandler()`.

11 Building LifeCycle Code

The `omniORB2` IDL compiler, `omniidl2`, only creates the additional stub code required for LifeCycle support if given the `'-l'` command-line switch. In addition to this, it is necessary to link with an extra library, `omniLC`. If you are using the `Omni` development environment, this is hidden away in the `OMNIORB2_IDL_LC_FLAGS` and `OMNIORB2_LC_LIB` makefile variables as shown in this excerpt from `dir.mk`:

```
OMNIORB2_IDL += $(OMNIORB2_IDL_LC_FLAGS)
...
$(lserver): lserver.o $(CORBA_STUB_OBJS) $(CORBA_LIB_DEPEND)
    @(libs="$(CORBA_LIB) $(OMNIORB2_LC_LIB)"; $(CXXExecutable))
```

12 Summary

In summary, these are the actions you need to take to use the LifeCycle support:

- Make sure you generate the LifeCycle support code by using `OMNIORB2_IDL_LC_FLAGS`.
- Derive your implementation class from the `_lc_sk_` skeleton.

- In your object factory, use `_this()` when returning a brand new object.
- When copying an object, call `_set_lifecycle()`, then use `_duplicate()` to return the object.
- To define your factory's copy operation in IDL, declare the `omniLifeCycleInfo` interface, but don't worry about its definition.
- To move an object create a copy of it, passing the result of `_get_lifecycle()` to the factory. Then call `_move()`.
- To remove an object, call the `_remove()` function.
- Implement concurrency control by deriving your implementation from `omniLC::_threadControl` and using the `omniLC::ThreadOp` and `omniLC::ThreadLC` classes.
- Maybe provide a new `CORBA::TRANSIENT` exception handler.
- Link your executables with the `omniLC` library.

13 How to use the examples

Run the `lcserver` program; it will output two stringified object references—the first to an `EchoLagFactory` and the second to an `EchoLag`. Now run `lcclient` with two arguments, the first a string to echo, and the second the `EchoLag` reference output by `lcserver`.

If all went well, `lcclient` will output the string `'First call'`. Further calls to `lcclient` will output the strings you give it.

Now, start another copy of `lcserver`. Run `lcclient` with three arguments—a string to echo, the `EchoLag` object reference of the first server, and the `EchoLagFactory` reference of the second server.

The object should be moved from the first server to the second, with the stored string maintained.

You can start more servers to experiment with moving objects from place to place. If you run `lcclient` with the `'-ORBtraceLevel 15'` argument, you will see the `TRANSIENT` exceptions being caught and handled.

When you get bored, run `lcremove` with the object reference of the original `EchoLag` to remove it. Further attempts to use `lcclient` or `lcremove` with that object will result in `OBJECT_NOT_EXIST` exceptions (or `COMM_FAILUREs` if you exit the servers).