Connections
ZeroC's Newsletter for the Ice Community
Issue Number 14, May 2006

Next generation grid computing

Grid computing is important, and it's here to stay. Without the ability to compose super computers from grids of cheap, ubiquitous hardware, solving many of the problems posed by the scientific community is simply too expensive and impractical. For example, the distributed.net project cracked RC5-64 after 1,757 days and 58,747,597,657 work units in 2002. Over the course of the project, 331,252 individuals participated and tested a total of 15,769,938,165,961,326,592 keys. Doing this type of computation on a single host is clearly infeasible.

Over the past couple of months, I've been evaluating various grid computing products. In general, they all work the same way. The user writes a job description, which identifies the executable, constraints (operating system, architecture, disk space, memory, etc), and associated input and output data. The user then submits the job to the grid and allows it to handle the details of selecting a free node, running the job, and retrieving the results.

By no means is this worthless. Without the grid software, users would have to do all of these steps themselves, which would be quite time consuming and frustrating. Furthermore, scheduling and allocating resources among multiple users is virtually impossible without a central authority. This type of grid computing platform is ideal for any application where the data can be easily subdivided among multiple workers, and the workers can work independently on their piece of the puzzle. However, this subset of problems is quite limiting, since there are many other types of problems that cannot be so easily decomposed, or that consist of a long work-flow where the data must be processed in several steps, or problems such as simulations where the results of one part of the simulation can potentially affect all other parts.

Solving these problems requires grids composed of interacting services. In my opinion, products that enable the construction of such grids must contain the following features: discovery, deployment, security, administration, data management, fault tolerance and scheduling. Together, these core features provide the infrastructure necessary to create grids of cooperating services. Furthermore, since the grid is a development tool and not a complete solution, it must also be lean, efficient, and most of all easy to learn, extend and use. With this in mind, I evaluated the current set of grid computing products, and found them all lacking. Most are moving in this direction, but at this time none have the features I feel are necessary.

I spent quite a bit of time evaluating the feature set of Globus, a popular grid computing product. It is interesting to compare the distributed computing infrastructures that Globus and IceGrid are using. As we have demonstrated on many occasions, Ice is a lean, mean, distributed computing platform. The protocol is small, fast and efficient. Globus has, to their detriment, selected web services as the basis of their distributed computing infrastructure. Much has been written by many people, including ZeroC's Michi Henning, on the poor performance and bandwidth utilization of web services. There is no better example of this than the gridftp service, a core utility used to transfer files around a Globus grid. Is this service written using web services? No. And for obvious reasons – the resulting bandwidth explosion would be enormous. It is telling that such a central Globus component cannot be written using their chosen distributed computing platform.

We at ZeroC strongly believe that grid computing is the key to solving big problems. We were very excited to release IceGrid as the major new feature of Ice 3.0, and we believe it is the key to the future growth and success of ZeroC and Ice. As such, we are continuing to work hard on IceGrid to add features essential to building the next generation of grid computing products. With Ice 3.1 you can look forward to enhanced scheduling and security for IceGrid.

[illegible]

Matthew Newhook
Senior Software Engineer

Issue Features

Integrating Ice with a GUI: Part III

In the third of a four part series on integrating Ice with a GUI, Matthew Newhook presents a non-invasive technique for making invocations without blocking the application.

The Samsara of Objects: Life Cycle Operations

Michi Henning discusses some of the issues involved in managing the life cycle of Ice objects.

Contents

Integrating Ice with a GUI: Part III

Matthew Newhook, Senior Software Engineer

Introduction

This is the third article in a series exploring the issues associated with using Ice in a graphical application. What we're striving to achieve is the ability to invoke an operation without blocking the calling thread, since a blocked thread can disrupt the user experience. The simple fact is any application thread that makes a remote invocation may find itself blocked until a timeout is reached. Of course, a regular twoway invocation always blocks the calling thread until the invocation is complete. However, oneway and asynchronous invocations, despite their non-blocking appearance, can also block the calling thread during connection establishment, or when underlying network buffers fill up. This article presents a solution for avoiding a blocked thread and requires only minimal changes to the application.

Request Interception

The technique used in the previous two columns requires the creation of a new subclass for each operation the application needs to invoke. An instance of this class is then pushed onto a work queue that makes the remote invocation from a separate thread. Since adding the call to the work queue cannot block, it is now possible to make oneway or asynchronous invocations without the risk of blocking the calling thread. Unfortunately, the need to create lots of small classes makes this a cumbersome technique to use. What I really want is a guarantee that the calling thread never blocks without giving up the traditional, more user-friendly invocation syntax shown in the example below:

```
// C++
HelloPrx hello = …;
hello->sayHello_async();
hello = HelloPrx::uncheckedCast(
    hello->ice_oneway());
hello->sayHello();
```

If we had a way to transparently capture the data associated with a request, we could push it onto a work queue and then make the actual invocation from a worker thread. The dynamic invocation and dispatch facility, commonly referred to as Dynamic Ice, provides most of what we need to implement this strategy. At this point, I recommend that you read Mark Spruiell's article on this subject from issue 11 of Connections.

First, let's briefly review how the `Blobject` class works. The class definition is summarized below:

```
// C++
class Blobject : virtual public Object
{
public:

    // Returns true if ok, false if user
    // exception.
    virtual bool ice_invoke(
        const std::vector<Byte>& inParams,
        std::vector<Byte>& outParams,
        const Current& current) = 0;
    // ...
};
```

By subclassing `Blobject`, the application can install a servant that intercepts the regular request-dispatching process and receives all invocations as calls to `ice_invoke`. The encoded input parameters are supplied by the `inParams` argument. A return value, output parameters, or a user exception must be encoded into the `outParams` argument. Finally, information about the request itself, such as the operation name and object identity, is contained in the `current` parameter. A return value of `true` indicates success, whereas `false` indicates a user exception.

A Blobject servant can easily forward a request to a proxy by calling the `ice_invoke` method defined by the base proxy class `ObjectPrx`:

```
// C++
class ObjectPrx : ...
{
public:

    bool ice_invoke(
        const string& name,
        OperationMode mode,
        const vector<Byte>& inParams,
        vector<Byte>& outParams);
    // ...
};
```

The return value has the same semantics as the Blobject `ice_invoke` method. In addition, there are versions of these methods that you can use to make asynchronous requests. An application can subclass `BlobjectAsync` to handle requests asynchronously:

```
// C++
class BlobjectAsync : virtual public Object
{
public:

    virtual void ice_invoke_async(
        const AMD_Object_ice_invokePtr& callback,
        const std::vector<Byte>& inParams,
        const Current& curr) = 0;
};
```

Similarly, `ObjectPrx` supplies an equivalent method for sending untyped invocations asynchronously:

```
// C++
class ObjectPrx : ...
{
public:

    void ice_invoke_async(
        const AMI_Object_ice_invokePtr& callback,
        const std::string& operation,
        OperationMode mode,
        const std::vector<Byte>& inParams,
        const Context& ctx);
    // ...
};
```

The asynchronous method invocation (AMI) operation `ice_invoke_async` expects the caller to supply a callback object in addition to the other information that comprises the request. The callback object must be a subclass of `AMI_Object_ice_invoke`, shown below:

```
// C++
class AMI_Object_ice_invoke : public ...
{
public:

    virtual void ice_response(bool ok,
        const vector<Byte>& results) = 0;
    virtual void ice_exception(
        const Exception& ex) = 0;
};
```

The Ice run time invokes `ice_response` if the request completes successfully or if a user exception occurred. The boolean argument has the value `true` for success and `false` for a user exception. The `results` argument supplies the encoded return values. In the case of a run time exception such as `ObjectNotExistException`, Ice calls `ice_exception` and passes the exception instance.

The technique that we'll explore in this article uses a Blobject servant to capture the request data. The servant encapsulates this data, along with the target proxy, and adds it to the call queue for eventual processing by a worker thread. To make this work, however, we must arrange for the application to direct its invocations to the local Blobject servant instead of the remote server. As a starting point, we'll require the application to register its target proxy with the Blobject servant, which returns an interposed proxy for the application to use instead.

Before we get started on the implementation, let's first examine the sequence of steps that must occur when making an invocation:

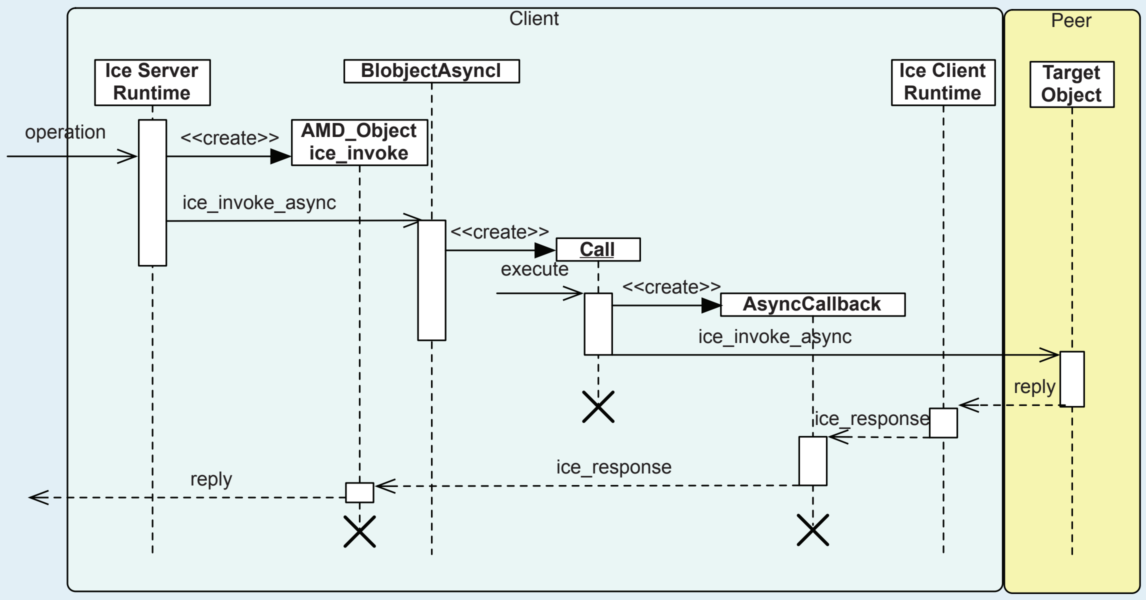
1. The application supplies the proxy for the target object and receives an interposed proxy to use in its place.
2. The application makes an invocation using the interposed proxy.
3. The Blobject servant intercepts the invocation.
4. The target proxy is looked up via the object identity.
5. The request data and target proxy are placed in a call object and added to the call queue.
6. The worker thread in the call queue executes the call by making the invocation on the target proxy.
7. The results of the invocation are returned to the application.

This all seems straightforward, except for the last part: how can we return the results of the invocation to the application? The trick here is to use request chaining, and to do that we need to use the asynchronous Blobject interface:

1. The Ice run time calls `ice_invoke_async` on the `BlobjectAsync` implementation.
2. The call object is created to hold the request data, the target proxy and the AMD callback object.
3. Upon execution, the call object uses `ice_invoke_async` on the target proxy to send the invocation. An AMI callback object is provided to the Ice run time.
4. The AMI callback relays the results to the AMD callback, which provides them to the application.

See Figure 1 for an interaction diagram of this process.

Figure 1: Interaction Diagram



Now that we've seen how the various objects interact, we'll begin our implementation with the Blobject servant. In addition to `ice_invoke_async`, which adds a new call to the queue, we also need a method (`add`) for registering a new target proxy. The resulting class definition is shown below:

```
// C++
class BlobjectAsyncI : public BlobjectAsync,
                        public Mutex
{
public:

    BlobjectAsyncI();

    virtual void
    ice_invoke_async(
        const AMD_Object_ice_invokePtr&,
        const vector<Byte>&,
        const Current&)
    void add(const ObjectPrx&);
    void destroy();

private:

    const CallQueuePtr _queue;
    map<Identity, ObjectPrx> _objects;
};
```

The constructor initializes and starts the call queue:

```
// C++
BlobjectAsyncI::BlobjectAsyncI() :
    _queue(new CallQueue)
{
    _queue->start();
}
```

The `add` method records a proxy in the `_objects` table:

```
// C++
void
BlobjectAsyncI::add(const ObjectPrx& proxy)
{
    Lock lock(*this);
    _objects.insert(make_pair(
        proxy->ice_getIdentity(),
        proxy->ice_newFacet("")->ice_twoway()));
}
```

The table key is the object identity, and the value is the proxy. The servant ensures that the proxy has no facet, and has semantics before adding it to the table. Without this step, an attempt to make an asynchronous invocation using the proxy might fail with `OnlyException`.

We've also defined a `destroy` method, which destroys the queue and joins with the queue's worker thread:

```
// C++
void
BlobjectAsyncI::destroy()
{
    Lock lock(*this);

    _queue->destroy();
    _queue->getThreadControl().join();
}
```

The request is intercepted by `ice_invoke_async`, whose implementation is straightforward:

```
// C++
void
BlobjectAsyncI::ice_invoke_async(
    const AMD_Object_ice_invokePtr& amdCallback,
    const vector<Byte>& inParams,
    const Current& curr)
{
    ObjectPrx proxy;
    {
        Lock lock(*this);
        map<Identity, ObjectPrx>::iterator p =
            _objects.find(curr.id);
        if(p != objects.end())
        {
            proxy = p->second;
        }
    }
    assert(proxy);
    _queue->add(new BlobjectCall(
        proxy, amdCallback, inParams, curr));
}
```

The method looks up the target proxy using the object identity provided by the `Current` argument, then enqueues a new call object to hold the request information.

Next we'll examine the implementation of `BlobjectCall`. The constructor is trivial and is omitted for the sake of brevity. More interesting is the `execute` method:

```
// C++
void
BlobjectCall::execute()
{
    ObjectPrx proxy = _proxy;
    if(!_curr.facet.empty())
    {
        proxy = proxy->ice_newFacet(_curr.facet);
    }
    proxy->ice_invoke_async(
        new AsyncCallback(_amdCallback),
        _curr.operation, _curr.mode, _inParams,
        _curr.ctx);
}
```

The method first obtains a proxy with the correct facet configuration, if necessary. Next, it sends the request asynchronously by calling `ice_invoke_async` on the target proxy. The `_amdCallback` member is supplied to the `AsyncCallback` class so that the reply can be sent back to the application. The implementation of the AMI callback simply relays the results to the AMD callback:

```
// C++
class AsyncCallback : public AMI_Object_ice_invoke
{
public:

    AsyncCallback(
        const AMD_Object_ice_invokePtr& cb) :
        _cb(cb)
    {
    }

    virtual void
    ice_response(bool ok,
            const vector<Byte>& results)
    {
        _cb->ice_response(ok, results);
    }

    virtual void
    ice_exception(const Exception& ex)
    {
        _cb->ice_exception(ex);
    }

private:

    const AMD_Object_ice_invokePtr _cb;
};
```

All of these internals now need to be exposed to the application. We'll wrap this up in a class called `SafeProxy`:

```
// C++
class SafeProxy : public Shared
{
public:

    SafeProxy(const CommunicatorPtr&);

    ObjectPrx add(const ObjectPrx&);
    void destroy();

private:

    const ObjectAdapterPtr _adapter;
    const BlobjectAsyncIPtr _blobject;
};
```

The constructor creates a special object adapter for use by the Blobject servant:

```
// C++
SafeProxy::SafeProxy(
    const CommunicatorPtr& communicator) :
    _adapter(communicator->
        createObjectAdapterWithEndpoints(
            "forward", "tcp -h 127.0.0.1")),
    _blobject(new BlobjectAsyncI)
{
    _adapter->activate();
}
```

The object adapter uses the loopback interface (by specifying `-h 127.0.0.1`), which prevents connections from other hosts. However, there is still a potential security problem: an application running on the same machine could connect to this object adapter and inject requests into the call queue. If this is a problem for your application, the configuration could be altered to use SSL.

Next, we'll look at the `add` method:

```
// C++
ObjectPrx
SafeProxy::add(const ObjectPrx& proxy)
{
    _blobject->add(proxy);
    return adapter->add(_blobject,
        proxy->ice_getIdentity())->
            ice_collocationOptimization(false);
}
```

The proxy is registered with the Blobject servant, and then the new Ice object is added to the adapter's active servant map. `ice_collocationOptimization` must be called since collocation optimization doesn't work with Blobject invocations.

Finally, the `destroy` method initiates the destruction of the Blobject servant:

```
// C++
void
SafeProxy::destroy()
{
    _blobject->destroy();
}
```

To put our plan into action, the application must call `SafeProxy::add` to obtain an interposed proxy. When that proxy is used for an invocation, the request is captured and queued by the Blobject servant. For example:

```
// C++
HelloPrx hello = …;
hello = HelloPrx::uncheckedCast(
    safeProxy->add(hello)->ice_oneway());
hello->sayHello();
```

The call to `sayHello` (or `sayHello_async`) will never block the calling thread as long as we use a oneway or asynchronous invocation. If you use a synchronous invocation, the calling thread will block while awaiting the reply – even if there are no return or out parameters. The example below illustrates this situation:

```
// C++
HelloPrx hello = …;
hello = HelloPrx::uncheckedCast(safeProxy-
>add(hello)->ice_());
hello->sayHello(); // Blocked until completion!
```

A downside to this technique is that it requires the application to manually call `SafeProxy:add` for each proxy. This requirement isn't much of a burden if the application uses just a few proxies – especially if they are known in advance. However, for applications that deal with a lot of proxies, this could become very tedious and lead to subtle issues that are difficult to diagnose and debug.

The Router Interface

Luckily, there is another implementation strategy that is completely transparent to the application. This solution utilizes the `Ice::Router` interface to eliminate the need for the application to explicitly register each proxy before using it.

If you have used Glacier2 in your applications, you will have encountered the router interface – although you may not have fully understood its purpose. Briefly, the router interface is used to redirect requests to another proxy. Once a proxy is configured with a router, the proxy's original endpoint is ignored, and all invocations made using that proxy are sent instead to the endpoint of the router. The router is responsible for forwarding the requests in an implementation-defined manner; Glacier2 is one example of a router implementation.

Although a router is typically a separate process, as is the case with Glacier2, there is nothing that prevents us from using a router in the same process. We'll take advantage of this fact and implement a router that adds requests to the call queue without blocking the calling thread.

The Ice run time supports the notion of a default router, which you can establish using the configuration property `Ice.Default.Router` or by calling `setDefaultRouter`, as shown below:

```
// C++
RouterPrx router = …;
communicator->setDefaultRouter(router);
ObjectPrx routedProxy =
    communicator->stringToProxy(...);
```

Defining the router in this fashion has a global effect: all proxies created subsequently are routed proxies by default. It is also possible to configure individual proxies with a router:

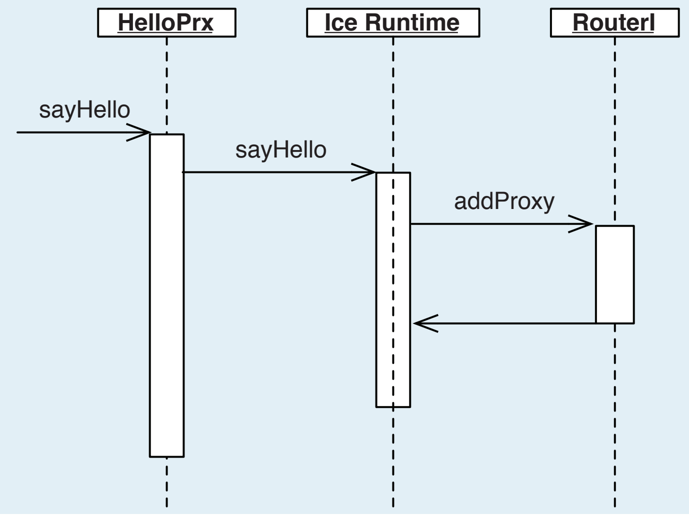
```
// C++
ObjectPrx obj = …:
RouterPrx router = …;
ObjectPrx routedProxy =
    obj->ice_router(router);
```

Any invocation on `routedProxy` is redirected to the router instead of being sent to the proxy's endpoint. Let us examine the `Ice::Router` interface in more detail:

```
// Slice
interface Router
{
    nonmutating Object* getClientProxy();
    nonmutating Object* getServerProxy();
    idempotent void addProxy(Object* proxy);
};
```

The Ice run time calls the `addProxy` operation during a proxy's initial invocation to register that proxy with its configured router. See Figure 2 for an interaction diagram.

Figure 2: Calling addProxy during initial invocation



Notice that the `addProxy` operation does not return a proxy. This differs from `SafeProxy::add`, where we needed to return a different proxy for the application to use in order to redirect invocations to our Blobject servant. Using the router interface instead, the Ice run time handles the forwarding details for us without forcing the application to jump through any hoops. Before it can forward a request, the Ice run time must call the `getClientProxy` operation to obtain the router's client-side proxy; all client invocations are sent to this proxy. The run time simply changes the identity of the client proxy before making the invocation. See Figure 3 for an interaction diagram of this process.

The `getServerProxy` operation is used by routed object adapters to set the endpoints of the proxies created by this adapter. For more details see the Ice manual. Since this application doesn't create any routed object adapters we don't need to implement this method.

The router interface is very similar to the `SafeProxy` class. The key difference is that the Ice run time already provides transparent support for routers; once the default router is established for the communicator, the application no longer needs to take any special actions to make use of the router.

Now that we've briefly reviewed the semantics of the router interface, let's take a look at one possible implementation:

```
// C++
class RouterI : public Router
{
public:

    RouterI(const CommunicatorPtr&);

    virtual ObjectPrx getClientProxy(
        const Current&) const;
    virtual ObjectPrx getServerProxy(
        const Current&) const;
    virtual void addProxy(
        const ObjectPrx&, const Current&);

    void destroy();

private:

    const ObjectAdapterPtr _adapter;
    const BlobjectAsyncIPtr _blobject;
    ObjectPrx _blobjectProxy;
};
```

The constructor performs all of the initialization work:

```
// C++
RouterI::RouterI(const CommunicatorPtr&
communicator) :
    _adapter(communicator->
        createObjectAdapterWithEndpoints(
            "forward", "tcp -h 127.0.0.1")),
    _blobject(new BlobjectAsyncI)
{
    _blobjectProxy = _adapter->addWithUUID(
        _blobject)->ice_collocationOptimization(
            false);

    RouterPrx proxy = RouterPrx::uncheckedCast(
        _adapter->addWithUUID(this)->
            ice_collocationOptimization(false));
    communicator->setDefaultRouter(proxy);
    _adapter->activate();
}
```

The constructor creates the object adapter and initializes a proxy that refers to the Blobject. As I mentioned earlier, the Blobject interface does not support collocation optimization, so this must be disabled in the proxy. Next, the constructor creates a proxy for the router and establishes it as the communicator's default router; any proxies created after this point will use this router. (Due to a bug in the Ice 3.0.1 run time, the router proxy also must not use collocation optimization.)

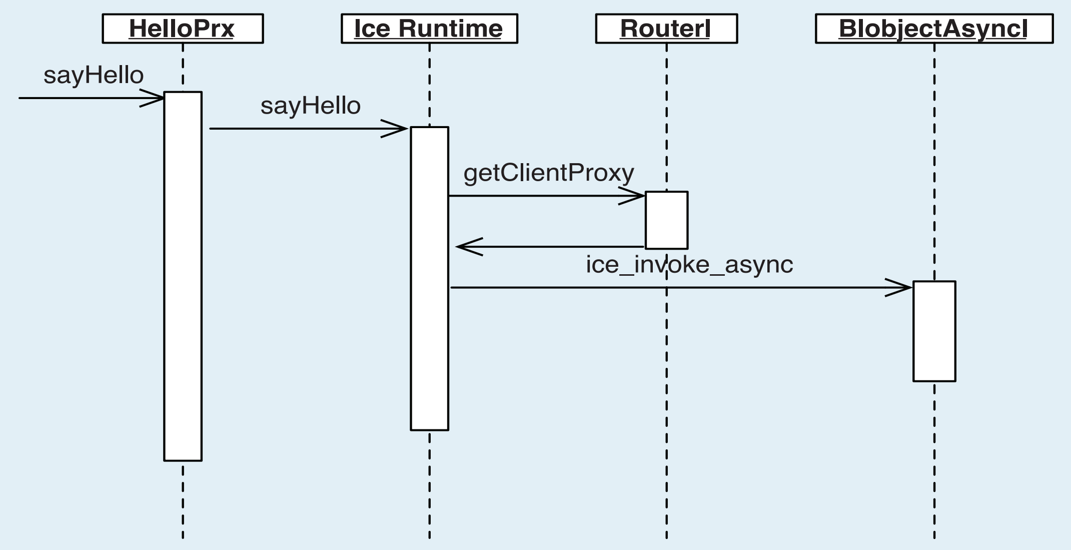
The remainder of the implementation is trivial. Note that it is not necessary to implement the `getServerProxy` method as previously mentioned

```
// C++
ObjectPrx
RouterI::getClientProxy(const Current&) const
{
    return _blobjectProxy;
}

ObjectPrx
RouterI::getServerProxy(const Current&) const
{
    assert(false); // Not implemented.
    return 0;
}

void
RouterI::addProxy(const ObjectPrx& proxy,
        const Current&)
{
    _blobject->add(proxy);
    _adapter->add(_blobject,
        proxy->ice_getIdentity());
}

void
RouterI::destroy()
{
    _blobject->destroy();
}
```

A further optimization can be applied: instead of adding each Ice object to the adapter's active servant map, we could use a servant locator. A servant locator is a callback interface that the adapter uses to locate servants that it can't find in its active servant map. Please consult the Ice manual for more information on this.

Using a servant locator, we can avoid adding the `_blobject` to the adapter's active servant map, and instead have the servant locator return `_blobject` for each request:

Figure 3: Use of getClientProxy by the Ice runtime



```
// C++
class ServantLocatorI : public ServantLocator
{
public:

    ServantLocatorI(
        const BlobjectAsyncIPtr& blobject) :
            _blobject(blobject)
    {
    }

    virtual ObjectPtr
    locate(const Current&, LocalObjectPtr&)
    {
        return _blobject;
    }


    virtual void
    finished(const Current&, const ObjectPtr&,
        const LocalObjectPtr&)
    {
    }

    virtual void
    deactivate(const string&)
    {
    }

private:

    const BlobjectAsyncIPtr _blobject;
};
```

To install the servant locator, we need to change our router's constructor:

```
// C++
RouterI::RouterI(const CommunicatorPtr&
communicator) :
    _adapter(communicator->
        createObjectAdapterWithEndpoints(
            "forward", "tcp -h 127.0.0.1")),
    _blobject(new BlobjectAsyncI)
{
    _adapter->addServantLocator(
        new ServantLocatorI(_blobject, "");
    //...
```

The implementation of `addProxy` can be simplified to the following:

```
// C++
void
RouterI::addProxy(const ObjectPrx& proxy,
    const Current&)
{
    _blobject->add(proxy);
}
```

One further change must be made that may not be immediately obvious. The current implementation of `BlobjectAsyncI::add` is repeated below:

```
// C++
void
BlobjectAsyncI::add(const ObjectPrx& proxy)
{
    Lock lock(*this);
    _objects.insert(make_pair(
        proxy->ice_getIdentity(),
        proxy->ice_newFacet("")->ice_twoway()));
));
}
```

Consider what would happen if the worker thread sent the invocation using this proxy: the Ice run time will examine the proxy, discover that it is configured with a router, and then forward the request to the router again! An infinite loop is clearly not our intent. To correct this problem, the router must be cleared from the proxy when adding it to the table:

```
// C++
void
BlobjectAsyncI::add(const ObjectPrx& proxy)
{
    Lock lock(*this);
    _objects.insert(make_pair(
        proxy->ice_getIdentity(),
        proxy->ice_newFacet("")->ice_twoway()->
        ice_router(0)));
}
```

If you want your application to use this technique in conjunction with Glacier2, you would call `ice_router` and pass the proxy for the Glacier2 router instead of a null proxy as we have done here. In this way, the worker thread's invocation would be forwarded automatically to Glacier2.

Oneway Invocations

How about oneway invocations? The code above always sends all requests as twoway invocations, but let's consider what happens if the application uses a oneway proxy as shown below:

```
// C++
HelloPrx hello = …;
hello = HelloPrx::uncheckedCast(
    hello->ice_oneway());
hello->sayHello();
```

The invocation of `sayHello` is sent as a oneway request to the Blobject servant, and then is forwarded as a twoway request to the actual target proxy. What if you really want the request to be sent as a oneway to the target proxy? Unfortunately, at the time of writing there is no way for a receiver to determine how a request was invoked, so some other method must be used.

Glacier2 addresses this limitation by looking for special directives in the request context. In particular, the value associated with the context key `_fwd` determines how Glacier2 forwards the request. For example, if the value is `o`, the request is forwarded as a oneway request. It would be quite simple to modify our router implementation to use the same scheme:

```
// C++
void
BlobjectCall::execute()
{
    ObjectPrx proxy = _proxy;
    if(!_curr.facet.empty())
    {
        proxy = proxy->ice_newFacet(_curr.facet);
    }
    Context::const_iterator p =
        _curr.ctx.find("_fwd");
    if(p != _curr.ctx.end() && p->second == "o")
    {
        proxy = proxy->ice_oneway();
        try
        {
            vector<Byte> out;
            bool ok = proxy->ice_invoke(
                _curr.operation, _curr.mode,
                _inParams, out, _curr.ctx);
            _amdCallback->ice_response(ok, out);
        }
        catch(const Exception& e)
        {
            _amdCallback->ice_exception(e);
        }
    }
    else
    {
        proxy->ice_invoke_async(
            new AsyncCallback(_amdCallback),
            _curr.operation, _curr.mode,
            _inParams, _curr.ctx);
    }
}
```

If you wanted to use multiple request queues, as shown in the previous article, then another possible implementation would be as follows:

```
// C++
void
BlobjectAsyncI::ice_invoke_async(
  const AMD_Object_ice_invokePtr& amdCallback,
  const vector<Byte>& inParams,
  const Current& curr)
{
      ObjectPrx proxy = ...;
      Context::const_iterator p =
            _curr.ctx.find("_fwd");
      if(p != _curr.ctx.end() &&
            p->second == "o")
      {
            CallPtr call =
                  new BlobjectOnewayCall(
                  proxy, amdCallback,
                  inParams, curr);
            _onewayQueue->add(call);
      }
      else
      {
            CallPtr call = new BlobjectCall(
                  proxy, amdCallback, inParams,
                  curr);
            _queue->add(call);
      }
}
```

Conclusion

I hope you'll find this technique useful. It does have an additional cost, since all request data is first sent over the loop-back interface and then copied into memory. This means that each invocation will have slightly higher latency, and will consume more memory in the client. However, it also means that the application's source code can be simpler and more concise, since it is no longer necessary to create lots of small call classes in order to send invocations without the fear of blocking.

The next article in this series will show you how to correctly handle return values from method invocations, as well as server-side invocations into a UI application.

The Samsara of Objects: Life Cycle Operations

Michi Henning, Chief Scientist

In *The Grim Reaper* (Issue 3 of *Connections*), I discussed how to use sessions to get rid of objects in a server if clients neglect to destroy the objects for some reason. That article tacitly assumed that the creation and destruction of Ice objects is already taken care of by the server. However, such life cycle operations tend to be fairly complex to implement, especially in the presence of threads. In this article, we will have a closer look at what actions a server must take to correctly create and destroy objects.

Creating Objects

The generic pattern for creating an Ice object is to provide a factory operation. For the purposes of this article, suppose that we are maintaining a collection of `Person` objects:

```
// Slice
interface Person
{
    nonmutating string name();
    nonmutating string getAddress();
    void setAddress(string addr);
};
```

Clearly, in a real application, these `Person` objects would be more complex and encapsulate a lot more state than just a person's name and address but, for this discussion, even such simple persons are sufficient to give us quite a bit to think about. The implementation of the person servant is very simple. (The methods are inline only to save space.)

```
// C++
class PersonI : public Person
{
public:

    PersonI(const string& name,
            const string& addr) :
        _name(name), _addr(addr)
    {
    }

    virtual string
    name(const Current&) const
    {
        return _name
    }

    virtual string
    getAddress(const Current&) const
    {
        return _addr;
    }

    virtual void
    setAddress(const string& addr,
               const Current&)
    {
        _addr = addr;
    }

private:
    string _name;
    string _addr;
};
```

For now, let us assume that no two persons can have the same name, so the name of a person object also acts as the object identity. (As we will see later, this might well be a bad idea; please bear with me regardless—it is instructive to consider the implications of this decision.) In order for clients to be able to create `Person` objects, we must provide a factory operation. The usual approach is to have the factory operation accept whatever state is necessary to initialize the object as parameters, and to return the proxy for the new object:

```
// Slice
exception PersonExists {};

interface PersonFactory
{
    Person* create(string name, string addr)
        throws PersonExists;
};
```

The job of `create` is to create a new person object, using the person's name as the object identity. Note that one immediate consequence of the decision to use a person's name as the object identity is that it introduces a potential error condition: a client might attempt to create a person with the same name as an already existing person, so the `create` operation must deal with this and throw a `PersonExists` exception in that case.

Also note that `create` has an `addr` parameter, so the state for the new person object can be initialized completely by `create`. As a general rule, factory operations should always initialize *all* of the state of the objects they create. You should avoid designs that create partially initialized objects that must be further initialized by making more calls. Not only is doing so less efficient, but it also creates a less robust design: if a programmer forgets to fully initialize an object and uses it, bad things tend to happen. By writing your factory operations such that they guarantee complete initialization of new objects, you eliminate this potential error at compile time.

By convention, factory operations return the proxy to the newly created object. However, this is by convention only. For example, the `create` operation could have `void` return type; in that case, clients could locate the `Person` object they have created by calling the `find` operation. Alternatively, you could define "bulk" factory operations that can create several objects in a single operation and return a proxy sequence to the created objects. Exactly how the factory operation works is entirely up to you—as far as the Ice run time is concerned, there is nothing special about a factory operation at all: a factory operation is like any other operation; it just so happens that the operation creates a new Ice object as a side effect of being called.

Here is a first cut at a class definition for the factory:

```
// C++
class PersonFactoryI : public PersonFactory
{
public:
    PersonFactoryI();
    virtual PersonPrx create(
        const string&, const string&,
        const Current&);
};
typedef Handle<PersonFactoryI>
    PersonFactoryIPtr;
```

Note that the factory is a singleton object, that is, only one instance of it ever exists in the server. As shown, the code does nothing to enforce this, but you can easily add code to the constructor that counts the number of factory instances and asserts or throws an exception if an attempt is made to instantiate a second instance.

Now let's see how we would implement the `create` operation. Here is a very simple version:

```
// C++
PersonPrx
PersonFactoryI::create(
    const string& name, const string& addr,
    const Current& c)
{
    try
    {
        return PersonPrx::uncheckedCast(
            c.adapter->add(
                new PersonI(name, addr),
                stringToIdentity(name)));
    {
    catch(const AlreadyRegisteredException&)
    {
        throw PersonExists();
    }
}
```

The `create` function instantiates a new `PersonI` servant, adds the servant to the active servant map (ASM), and returns the proxy to the newly created Ice object. As far as the Ice run time is concerned, adding the servant to the ASM is what creates a new Ice object: as soon as a new entry appears in the ASM, client requests for the corresponding object identity will be dispatched to the servant that is registered for that identity in the ASM.

Note that, even though no locking is involved, this code is thread-safe. The `add` operation behaves atomically, that is, if two clients attempt to concurrently create the same person, so two threads are active in the body of `create`, exactly one of the two threads will succeed in adding the servant to the ASM; the other thread will receive an `AlreadyRegisteredException`. On the other hand, if two clients attempt to concurrently create two different persons, they can proceed in parallel without danger of corrupting program state. (Internally, in the implementation of `add`, the two threads are serialized by the Ice run time while they access the ASM.) The constructor of the `PersonI` servant does nothing, so no lock is required inside that constructor either. However, for a more complex `PersonI` implementation, the constructor might update a shared data structure, in which case we would have to add appropriate locking to protect that data structure.

Also note that the preceding code does not contain a memory leak. It is fine to pass the pointer returned by `new` directly to `add` because the formal parameter type of `add` is `const Ice::ObjectPtr&`, and `ObjectPtr` has a constructor that accepts an `Object*`. This means that the ASM stores a smart pointer to the servant. Because servants are reference counted, the code contains no memory leak; the servant will be deallocated once its ASM entry is removed.

Destroying Objects

Now that we can create `Person` objects, let us consider how to destroy them again. Here is one possible approach, namely, adding a `destroy` operation to the `PersonFactory`:

```
// Slice
exception PersonExists {};
exception PersonNotExists {};

sequence<Person*> PersonSeq;

interface PersonFactory
{
    Person* create(string name, string addr)
        throws PersonExists;
    void destroy(Person* p)
        throws PersonNotExists; // Bad idea!
};
```

At first glance, this looks reasonable. Seeing that clients go to the `Person` factory to create a person, it seems only logical that they go back to the same factory to destroy that person: after all, the factory knows how to create persons, so surely it also knows how to destroy them again.

Unfortunately, this is generally a bad idea. For one, it is possible that a client may attempt to destroy a person that was already destroyed earlier. Our Slice definition has to explicitly cater for this possibility with a `PersonNotExists` exception, which makes the interface just that little bit more complex. Second, and more importantly, with this design, to destroy a person, a client must not only know which person to destroy, but must also know *which factory created that person*. This may not sound like a big deal—however, in large and complex systems with dozens of factories (and possibly multiple person factories in different servers), this rapidly becomes a problem: for each object, the application code somehow has to keep track of which factory created that object; if any part of the code ever loses track of where an object originally came from, it can no longer destroy that object. Of course, we could mitigate the problem by adding an operation to the `Person` interface that returns a proxy to the `PersonFactory` that created the person. Clients could then ask a person for the factory proxy so they could then call the `destroy` operation on the factory. But that would make the Slice definitions yet more complex and really is just a band-aid on a fundamentally flawed design. A much better choice is to add the `destroy` operation to the `Person` interface instead:

```
// Slice
interface Person
{
    // ...
    void destroy();
};
```

With this approach, there is no need for clients to somehow keep track of which factory created what person. Instead, given a proxy to a `Person` object, a client simply invokes the `destroy` operation on the object and the person obligingly commits suicide. In addition, there is no need to define an additional `PersonNotExists` exception: if the person was previously destroyed by another client, any call to `destroy` raises `ObjectNotExistException` (which exists precisely to indicate this condition).

So, how can we implement `destroy`? As far as the Ice run time is concerned, what destroys an Ice object is the act of removing the object's ASM entry. Doing this breaks the link between the object identity and the servant, and any incoming requests for that object once the ASM entry is removed raise `ObjectNotExistException`. Removing the ASM entry for a servant will cause the servant to be destroyed once the last operation that is still executing inside the servant completes. This happens because servants are reference counted by the Ice run time. As long as a servant has an ASM entry, its reference count is one (assuming that your program does not hold any other smart pointers to the servant elsewhere). In addition, the Ice run time increments the servant's reference count by one for each currently executing operation. This means that, if you remove the ASM entry for a servant while operations are executing inside the servant, the servant's reference count drops by one, but does not reach zero immediately. Instead, as each executing operation completes, the Ice run time decrements the servant's reference count until the count finally reaches zero, namely, when the last executing operation completes. At that point, the run time invokes the servant's destructor. This leads to an implementation of `destroy` as follows:

```
// C++
void
PersonI::destroy(const Current& c)
{
    try
    {
        c.adapter->remove(c.id);
    }
    catch(const NotRegisteredException&)
    {
        throw ObjectNotExistException(
            __FILE__, __LINE__);
    }
}
```

Note the try–catch block around the call to `remove`. Catching and handling `NotRegisteredException` is necessary if two clients concurrently invoke `destroy` on the same object. In that case, only one thread will succeed in removing the ASM entry for the servant, and the other thread will receive a `NotRegisteredException`. If we did not catch and handle that exception, the Ice run time would translate it into an `UnknownLocalException` and the client that invoked the `destroy` operation would conclude that something is internally broken in the server. By handling the exception, we ensure that the client gets `ObjectNotExistException`, as it should.

Because smart pointers nicely ensure that a servant is destroyed only once it becomes idle (that is, once the last executing invocation drains out of the servant), we need not do anything else in `destroy`. (Of course, this assumes that no other part of the program holds a smart pointer to the same servant, which would prevent the servant's reference count from reaching zero.)

It is important to be aware of the distinction between an Ice object and its servant: the Ice object is an abstraction and, as far as the run time is concerned, existence of an Ice object constitutes the ability to successfully dispatch a request to the servant: if an ASM entry exists, the Ice object exists.[1] On the other hand, the servant is simply the physical manifestation of the abstract Ice object, and the servant can exist in isolation of its Ice object. The life cycles of Ice objects and servants are completely separate.

Cleaning Up Servant State

As shown, our `destroy` operation does exactly the right thing because the `PersonI` destructor only performs trivial actions. (The implicit destructor simply invokes the destructors of the `_name` and `_addr` members.) However, in a realistic application, cleaning up servant state is often not that simple. In particular, clean-up can involve complex actions, such as committing a transaction, updating the file system, or making a remote call on another object. For example, our `PersonI` servants could be implemented by storing their state in a file; when a `PersonI` servant is destroyed, the destructor could close an `fstream` for that file:

```
// C++
PersonI::~PersonI()
{
    _myStream.close(); // Bad idea
}
```

1. I'm ignoring servant locators here. However, that does not substantially alter the discussion: if you use a servant locator, the existence of the Ice object is determined by whether the servant locator's `locate` operation returns a servant or not, that is, the servant locator effectively replaces the ASM.

The problem with this clean-up code is that it can fail, for example, if the file system is full and buffered data cannot be written to the file. Such clean-up failure is a general issue for non-trivial servants: for example, a transaction can fail to commit, an update to a file system or database can fail, and a remote call can fail if the network goes down.

If we attempt to clean up such servant state from within the servant's destructor and something goes wrong, we have a serious problem: we cannot inform the client of the error because, as far as the client is concerned, the `destroy` call completed just fine. The client will therefore assume that the servant was correctly destroyed. However, the system is now in an inconsistent state: the Ice object for the person has been destroyed (because its ASM entry was removed), but the person's state still exists (possibly with incorrect values). This can cause errors later.

Another reason for avoiding state clean-up in C++ destructors is that destructors cannot throw exceptions: if they do, and do so in the process of being called during unwinding of the stack due to some other exception, the program goes directly to `terminate` and does not pass "Go". (There are a few exotic cases in which it is possible to throw from a destructor and get away with it but, in general, it is an excellent idea to maintain the no-throw guarantee for destructors.) So, if anything goes wrong during destruction, we are in a tight spot: we are forced to swallow any exception that might be encountered by the destructor, and the best we can do is log the error, but not report it to the client.

Finally, using destructors to clean up servant state does not port well to languages such as Java and C#. For these languages, similar considerations apply to error reporting from a finalizer and, with Java, finalizers may not run at all. So, in general, I recommend that you perform any clean-up actions in the body of `destroy` instead of delaying clean-up until the servant's destructor runs. Note that the foregoing does *not* mean that you cannot reclaim servant resources in destructors; after all that is what destructors are for. But it *does* mean that you should not try to reclaim resources from a destructor if the attempt can fail (such as deleting records in an external system as opposed to, for example, deallocating memory).

Life Cycle and Collection Operations

At the moment, our factory is what is known as a *pure* factory: it only provides a `create` operation. However, it is common for factories to double up as collection managers and to provide `find` and `list` operations:

```
// Slice
exception PersonExists {};

sequence<Person*> PersonSeq;

interface PersonFactory
{
    Person* create(string name)
        throws PersonExists;
    nonmutating Person* find(string name);
    nonmutating PersonSeq list();
};
```

`find` returns a proxy for the person with the specified name, and a null proxy if no such person exists. `list` returns a sequence that contains the proxies of all existing persons.

The implementation of `find` is very simple:

```
// C++
PersonPrx
PersonFactoryI::find(const string& name,
                     const Current& c)
{
    PersonPrx p;
    Identity id =
        stringToIdentity(name);
    if(c.adapter->find(id))
    {
        p = PersonPrx::uncheckedCast(
                c.adapter->createProxy(id));
    }
    return p;
}
```

The object adapter's `find` operation returns the smart pointer to the servant if a corresponding entry exists in the ASM, and a null pointer otherwise. Again, there are no threading issues in this code because the Ice run time ensures that ASM accesses are interlocked. If another client concurrently invokes `create` or `destroy` while `find` is running, the right thing happens automatically: `find` reports the current state of affairs without race conditions.

Cyclic Dependencies

When it comes to implementing `list`, we hit a snag: `list` needs to iterate over the collection of persons and return the proxy for each person, but the object adapter does not provide any iterator for ASM entries. There is a good reason for this: during iteration, the ASM would have to be locked to protect it against concurrent modification, but locking the ASM would prevent call dispatch and possibly create deadlocks. Therefore, we cannot use the ASM for iteration and must maintain our own list of existing persons and use it to implement the `list` operation:

```
// C++
class PersonFactoryI : public PersonFactory
{
public:
    // ...
    void _remove(const string&
                 const ObjectAdapterPtr&);
private:
    Mutex _lcMutex;
    set<string> _persons;
};
```

The factory maintains a set of strings that stores the name of each person. The implementation of `list` iterates over the `_persons` set to create a proxy for each person, and `create` and `destroy` update the set of names appropriately. However, before we launch into the implementation, we need to think about concurrency. For example, it is possible for one client to call `create` or `destroy` while `list` is running. Clearly, we must interlock `create`, `destroy`, and `list`—if we do not, and two threads concurrently modify the `_persons` set, or a thread modifes the set while `list` iterates over it, we run the risk of a crash because STL containers are not thread-safe. We can deal with this issue by adding a member variable `_lcMutex` (*l*ife *c*ycle *mutex*) to the factory. This mutex can be locked by `create`, `destroy`, and `list` to prevent concurrent access to the set of persons.

We also have another issue with `destroy`: when a person is destroyed, we must remove the corresponding entry from the factory's set of persons, so `destroy` must somehow inform the factory when a person disappears. This is the purpose of the `_remove` method: it removes the specified name from the `_persons` set and removes the servant's ASM entry under protection of the `_lcMutex` lock. Of course, for `destroy` to be able to call `_remove`, it must hold a smart pointer to the factory. Seeing that the factory is a singleton, we can do this by adding a static `_factory` member to our `PersonI` class:

```
// C++
class PersonI : public Person
{
public:
    // ...
    static PersonFactoryIPtr _factory;
private:
    string _name;
    string _addr;
};
```

The code in `main` then creates the factory and initializes the static member variable, for example:

```
// C++
PersonI::_factory = new PersonFactoryI();
// Add factory to ASM and activate object
// adapter...
```

All this is fine as far as it goes, but it leaves a bad taste in our mouth because it sets up a cyclic dependency: the factory class knows about the person class, and the person class knows about the factory so it can call `_remove`. In general, such cyclic dependencies are a bad idea: if nothing else, they make a design harder to understand (and are possibly an indication of sloppiness).

We could remove the cyclic dependency by moving the person set and its associated mutex into a separate class instance that is referenced from both `PersonFactoryI` and `PersonI`. That would get rid of the cyclic dependency as far as the C++ type system is concerned but, as we will see later, it would not really help because the factory and the servants turn out to be mutually dependent regardless. So, for the moment, I will stay with this design, simply so we can look at the threading issues around life cycle and collection manager operations. Once we have covered the basic ideas, I will present an alternative design that solves the cyclic dependency problem much more elegantly.

Implementation

With the design as it stands, `list` can be implemented as follows:

```
// C++
PersonSeq
PersonFactoryI::list(const Current& c) const
{
    Mutex::Lock lock(_lcMutex);

    PersonSeq ps;
    set<string>::const_iterator i;
    for(i = _persons.begin(); i != _persons.end();
        ++i)
    {
        ps.push_back(PersonPrx::uncheckedCast(
            c.adapter->createProxy(
                stringToIdentity(*i))));
    }
    return ps;
}
```

Note that `list` acquires a lock on the life cycle mutex, to prevent concurrent modification by `create` and `destroy`. In turn, our `create` implementation now also locks the life cycle mutex:

```
// C++
PersonPrx
PersonFactoryI::create(const string& name,
                       const string& addr,
                       const Current& c)
{
    Mutex::Lock lock(_lcMutex);
    PersonPrx p;
    try
    {
        p = PersonPrx::uncheckedCast(
            c.adapter->add(
                new PersonI(name, addr),
                   stringToIdentity(name)));
    }
    catch(const AlreadyRegisteredException&)
    {
        throw PersonExists();
    }
    _persons.insert(name);
    return p;
}
```

This implementation is safe in the face of concurrent execution of `list`: only one of `create` and `list` can access the person set at a time, so there are no race conditions.

The implementation of `destroy` similarly acquires the life cycle mutex, by calling `_remove` on the factory:

```
// C++
void
PersonI::destroy(const Current& c)
{
    _factory->_remove(_name, c.adapter);
}
```

The `_remove` implementation locks `_lcMutex`, removes the ASM entry, and deletes the person's name from the `_persons` set:

```
void
PersonFactoryI::_remove(
    const string& name,
    const ObjectAdapterPtr& a)
{
    Mutex::Lock lock(_lcMutex);
    try
    {
        a->remove(stringToIdentity(name));
    }
    catch(const NotRegisteredException&)
    {
        throw ObjectNotExistException(
            __FILE__, __LINE__);
    }
    _persons.erase(name);
}
```

With this implementation, `create`, `list`, and `destroy` are correctly interlocked. Only one of the three operations can proceed at a time, so we can be sure not to corrupt the `_persons` set with concurrent updates.

Life Cycle and Normal Operations

So far, we have mostly ignored the implementation of the `name`, `getAddress`, and `setAddress` operations. Obviously, `getAddress` and `setAddress` must be interlocked against concurrent access, otherwise one client could modify the address while another client is reading it. To deal with this, we can add a mutex `_m` to `PersonI`:

```
// C++
class PersonI : public Person
{
public:
    // ...
private:
    string _name;
    string _addr;
    Mutex _m;
};

string
PersonI::getAddress(const Current&) const
{
    Mutex::Lock lock(_m);
    return _addr;
}
```

```
void
PersonI::setAddress(const string& addr,
    const Current&)
{
    Mutex::Lock lock(_m);
    _addr = addr;
}
```

This looks good but, as you might have expected, the presence of `destroy` throws a spanner into the works: as shown, this code suffers from a rare, but real, race condition. Consider the situation where a client calls `destroy` at the same time as another client calls `setAddress`. The two calls are dispatched in separate threads and can therefore proceed concurrently.

The following sequence of events can occur:

1. The thread dispatching the `setAddress` call locates the servant, enters the operation implementation, and is suspended by the scheduler immediately on entry, before it executes any of the statements in the body of the operation.
2. The thread dispatching the `destroy` call locates the servant, enters `destroy`, acquires the life cycle lock, successfully removes the servant from the person set and the ASM, and returns.
3. The thread calling `setAddress` is scheduled again, acquires the lock, and now operates on a conceptually already-destroyed Ice object.

The problem here is that a thread can enter the servant and be suspended before it gets a chance to acquire a lock. With the code as it stands, this is not a problem: `setAddress` will simply update the address of a servant that no longer has an ASM entry. In other words, the Ice object is already destroyed—it's just that the servant is still hanging around because there are still operations executing inside it. Any updates to that servant will succeed (even though they are useless because the servant's destructor will run as soon as the last operation leaves the servant).

While this race condition does not affect our implementation, it *does* affect more complex applications, particularly if the application modifies external state, such as a file system or database. For example, `setAddress` could modify a file in the file system; in that case, `destroy` would delete that file and probably close a file descriptor. If we were to allow `setAddress` to execute after `destroy` has already done its job, we would likely encounter problems: either `setAddress` would not find the file where it expects it to be or try and to use the closed file descriptor and return with an error or, worse, `setAddress` could end up re-creating the file in the process of updating the already destroyed person's address. (Of course, we can handle this error condition but, for systems with more complex servant implementations, dealing with such errors can be more difficult.)

This scenario illustrates a general issue for applications that allow clients to destroy objects: we must consider, for each operation on a servant, whether concurrent destruction of that servant can cause the operation to fail or to corrupt system state. If so, we must make sure that operations such as setAddress notice when the object has been destroyed previously and throw an ObjectNotExistException. We can do this by adding a _destroyed member to the PersonI servant. This member is initialized to false by the constructor and set to true by destroy. On entry to every operation (including destroy), we lock the mutex, test the _destroyed flag, and throw ObjectNotExistException if the flag is set:

```
// C++
class PersonI : public Person
{
public:
    // ...
private:
    string _name;
    string _addr;
    bool _destroyed;
    Mutex _m;
};

PersonI::PersonI(const string& name
                 const string& addr) :
    name(_name), _addr(addr), _destroyed(false)
{
}

string
PersonI::name(const Current&) const
{
    Mutex::Lock lock(_m);
    if(_destroyed)
    {
        throw ObjectNotExistException(
            __FILE__, __LINE__);
    }
    return _name;
}

string
PersonI::getAddress(const Current&) const
{
    Mutex::Lock lock(_m);
    if(_destroyed)
    {
        throw ObjectNotExistException(
            __FILE__, __LINE__);
    }
    return _addr;
}

void
PersonI::setAddress(const string& addr, const
Current&)
{
    Mutex::Lock lock(_m);
    if(_destroyed)
    {
        throw ObjectNotExistException(
            __FILE__, __LINE__);
    }
    _addr = addr;
}

void
PersonI::destroy(const Current& c)
{
    Mutex::Lock lock(_m); // Dubious!
    if(_destroyed)
    {
        throw ObjectNotExistException(
            __FILE__, __LINE__);
    }
    _destroyed = true;
    _factory->_remove(_name, c.adapter);
}
```

If you are concerned about the repeated code to test the _destroyed flag and throw ObjectNotExistException on entry to each operation, you can bundle that code into a member function or base class to make it reusable. (I'll leave this an exercise.)

With this implementation, if an operation is dispatched, but suspended before its body executes and, meanwhile, destroy runs to completion, it becomes impossible for an operation to operate on the state of a "zombie" servant: the test on entry to each operation ensures that any operation that runs after destroy is immediately "thrown out."

Also note the "dubious" comment in destroy: the operation first acquires the mutex _m and, while holding that mutex, calls _remove, which attempts to lock the life cycle lock. This is not wrong as such but, as we will see in a moment, it can easily lead to deadlock if the application is modified later.

Removing Cyclic Dependencies

Earlier, I said that factoring the person set and its mutex into a separate class instance would not really solve the cyclic dependency problem (at least not in general). To see why, suppose that we want to expand our factory with a new getDetails operation:

```
// Slice
struct PersonDetails
{
    Person* proxy;
    string name;
    string address;
};
sequence<PersonDetails> PersonDetailsSeq;

interface PersonFactory
{
    // ...
    PersonDetailsSeq getDetails();
};
```

This kind of operation is common in collection managers: instead of returning a simple list of proxies, getDetails returns a sequence of structures, each of which contains not only the object's proxy, but also some of the state of the corresponding object. The motivation for this is performance: with a plain list of proxies, the client, once it has obtained the list, is likely to immediately follow up with one or more remote calls to the objects in the list in order to retrieve their state, which is inefficient.

To implement getDetails, we need to iterate over the set of persons and invoke the name and getAddress operation on each person to retrieve the state that is to go into each PersonDetails structure. But that turns out to be rather dangerous because the following sequence of events is possible:

- Client A calls getDetails.
- The implementation of getDetails must lock the life cycle mutex to ensure that the set of persons cannot change while it iterates over the set.
- Client B calls destroy on an object.
- The implementation of destroy locks the object's mutex _m, sets the _destroyed flag, and then calls _remove, which attempts to lock _lcMutex. In turn, _lcMutex is already locked by getDetails, so _remove blocks until _lcMutex is unlocked.
- If getDetails now invokes the getAddress operation on the same object, getAddress will attempt to lock that object's mutex _m.

At this point, everything stops, because each operation holds a lock on a mutex while it tries to acquire another mutex that is already locked by the other operation.

There are two basic ways to address this problem:

- Change the locking such that the deadlock becomes impossible.
- Abandon the idea of calling back from servants into the collection manager, and use a reaping strategy instead.

Deadlock-Free Lock Acquisition

The first option is fairly easy to implement for our example: in destroy, we set the _destroyed flag while _m is locked, and then unlock _m again before calling back into the factory:

```
void
PersonI::destroy(const Current& c)
{
    {
        Mutex::Lock lock(_m);
        if(_destroyed)
        {
            throw ObjectNotExistException(
                __FILE__, __LINE__);
        }
        _destroyed = true;
    }
    _factory->_remove(_name, c.adapter);
}
```

This fixes the problem because destroy releases _m before acquiring _lcMutex, so any operation invocations made by getDetails do not deadlock on _m. But there is a more serious issue with this approach: for a more complex application, rearranging locking in this fashion may be quite difficult. If callbacks involve several objects and do complex things, it can be next to impossible to prove that the code is free of deadlocks. This is particularly true if the code uses condition variables and suspends threads until a condition becomes true. (You may want to check out Bernard Normier's articles in Issue 4 and Issue 5 of *Connections*, which examine strategies for dealing with deadlocks in detail.)

At the core of the problem is that concurrency can create circular locking dependencies: an operation on the factory can require the same locks as a concurrent call to destroy. This is one reason why threaded code is harder to write than sequential code—the semantic interactions among operations require locks, but dependencies among the various locks are not immediately obvious. In effect, locks set up an entirely separate and largely invisible set of dependencies: it was easy to spot the mutual dependency between PersonFactoryI and PersonI because the dependency was manifest in the type system; in contrast, it was much harder to spot the lurking deadlock in the implementation of destroy. And, worse, the deadlock may never be found in testing and only show up only once the application is deployed.

Servant Reaping

Rather than going to great lengths to ensure that no deadlocks are possible in the face of concurrent execution of normal operations, collection manager operations, and destroy, it is often better to change to a reaping strategy: instead of updating the factory's set of persons immediately, destroy marks a servant as destroyed and removes its ASM entry, and the factory deals with updating the set of servants at a more convenient time. Here is an outline of this approach:

- destroy marks the servant as destroyed and removes the ASM entry as usual, but does not call back into the factory to update the set of persons.
- Every time create or list are called, the factory scans the set of servants and removes any servants that have been destroyed.

This makes for a much cleaner design: it avoids both the cyclic type dependency and the cyclic locking dependency.

To implement reaping, we need to change the definition of PersonI. It no longer has a static smart pointer to the factory but now provides a member function _isZombie that allows the factory to check the _destroyed flag of a servant:

```
// C++
class PersonI : public Person
{
public:
    PersonI(const string&,
            const string&);
    virtual string name(
            const Current&) const;
    virtual string getAddress(
        const Current&) const;
    virtual void setAddress(
        const string&, const Current&);
    virtual void destroy(const Current&);
    bool _isZombie() const;

private:
    string _name;
    string _addr;
    Mutex _m;
    bool _destroyed;
};
typedef Handle<PersonI> PersonIPtr;
```

The implementation of `_isZombie` is trivial: it returns the `_destroyed` flag under protection of the lock:

```
// C++
bool
PersonI::_isZombie() const
{
    Mutex::Lock lock(_m);
    return _destroyed;
}
```

The implementation of `destroy` no longer calls back into the factory to update the person set. Instead, it simply sets the `_destroyed` flag and removes the servant's ASM entry:

```
void
PersonI::destroy(const Current& c)
{
    Mutex::Lock lock(_m);
    if(_destroyed)
    {
        throw ObjectNotExistException(
            __FILE__, __LINE__);
    }
    _destroyed = true;
    c.adapter->remove(c.id);
}
```

The `PersonFactoryI` class now stores a map of name–servant pairs instead of a set of names:

```
// C++
class PersonFactoryI : public PersonFactory
{
public:
    // Constructor and Slice operations here...

private:
    typedef map<string, PersonIPtr> PMap;
    mutable Pmap _persons;
    Mutex _lcMutex;
};
```

The implementation of `create` illustrates how reaping works. As for the previous implementation, `create` first instantiates a servant and attempts to adds it to the ASM. But, before adding the servant to the `_persons` map, it scans the map for zombies and removes them:

```
// C++
PersonPrx
PersonFactoryI::create(const string& name,
                       const string& addr,
                       const Current& c)
{
    Mutex::Lock lock(_lcMutex);
    PersonPrx p;
    PersonIPtr servant;
    try
    {
        servant = new PersonI(name, addr);
        p = PersonPrx::uncheckedCast(
            c.adapter->add(
                servant, stringToIdentity(name)));
    }
    catch(const AlreadyRegisteredException&)
    {
        throw PersonExists();
    }
    PMap::iterator i = _persons.begin();
    while(i != _persons.end())
    {
        if(i->second->_isZombie())
        {
            _persons.erase(i++);
        }
        else
        {
            ++i;
        }
    }
    _persons[name] = servant;
    return p;
}
```

The implementation of `list` similarly scans the map for zombie servants. Because `list` needs to iterate over the map anyway, reaping incurs essentially no extra cost:

```
PersonSeq
PersonFactoryI::list(const Current& c) const
{
    Mutex::Lock lock(_lcMutex);
    PersonSeq ps;
    PMap::iterator i = _persons.begin();
    while(i != _persons.end())
    {
        if(i->second->_isZombie())
        {
            _persons.erase(i++);
            }
            else
            {
                ps.push_back(PersonPrx::uncheckedCast(
                    c.adapter->createProxy(
                        stringToIdentity(i->first))));
                ++i;
            }
        }
        return ps;
    }
```

With this approach, we have a much cleaner design: there is no cyclic dependency (either explicit, in the type system, or implicit, as a locking dependency) between the factory and the servants. Moreover, the implementation is easier to understand, once you get used to the idea of reaping: there is no need to follow complex callbacks and to carefully analyze the order of lock acquisition. In general, for all but the most trivial applications, reaping is therefore a better approach than calling back from the servants into the factory.

Alternative Reaping Implementations

One thing you might be concerned about is that, with reaping, the cost of `create` has increased from $O(\log n)$ to $O(n)$ because `create` now iterates over the map of persons and locks and unlocks every servant in the map. Generally, this is not an issue because life cycle operations are called infrequently compared to normal operations. You will notice this additional cost only if you have a very large number of servants (in the tens of thousands at least) *and* life cycle operations are called very frequently.

If you find that `create` is a bottleneck (by profiling the application, not by guessing!), you can change to a more efficient implementation by adding zombie servants to a separate set. In that case, reaping iterates over the zombie set instead of the main map and removes each servant that is in the zombie set from the main map. With this change, `create` is still an $O(n)$ operation, but the cost is reduced to being proportional to the number of *zombie* servants, instead of the *total* number of servants. In addition, we can lock and unlock just once, instead doing it n times.

You may also be concerned about the number of zombie servants that can accumulate in the server if `create` is not called for some time. Again, for most applications, this is not a problem: the servants occupy memory, but no other resources because `destroy` can still clean up scarce resources, such as network connections and file descriptors. If you really need to prevent accumulation of zombie servants (again, determined by profiling, not by guessing!), you can reap from a background thread that runs periodically, or you can count the number of zombies and trigger a reaping pass once that number exceeds some threshold.

Life Cycle and Parallelism

When we look at the degree of concurrency that is supported by our application, we find:

- All operations on the factory are serialized, so only one of `create`, `list`, and `find` can execute at a time.
- Concurrent operation invocations on the same servant are serialized, but concurrent operation invocations on different servants can proceed in parallel.

For the vast majority of applications, this is entirely adequate: life cycle operations are rare compared to normal operations, as are concurrent invocations on the same servant. However, for some applications, serializing `list` and `find` can be a problem, particularly if they are implemented by iterating over a large number of records in a collection of files or a database. In that case, the operation might take quite a while to complete. Moreover, `list` and `find` do not actually change any application-visible state so, on the face of it, there is no reason to prevent clients from executing these operations concurrently.

If you find that you need the extra concurrency, you can interlock `create`, `list`, and `find` by using a read–write recursive mutex. Such a mutex provides separate operations for acquiring a read lock and a write lock. Multiple readers can concurrently hold a read lock, but a write lock requires exclusive access: the write lock is granted to exactly one writer only once there are no readers or writers holding the lock. If a writer is waiting to get a write lock, readers attempting to get a read lock are delayed, that is, writers are given preference and get hold of the write lock as soon as the last current reader releases its lock.

Ice provides a read–write recursive mutex with the `IceUtil::RWRecMutex` class. We can gain increased parallelism by changing the type of `_lcMutex` to `RWRecMutex`. `create` then acquires a write lock on this mutex, and `list` and `find` acquire a read lock. This allows calls to `list` and `find` to proceed concurrently, but `create` can run only while no calls to `list` and `find`, and no other calls to `create` are in progress:

```
// C++
class PersonFactoryI : public PersonFactory
{
public:
    // ...

private:
    // ...
    RWRecMutex _lcMutex;
};

PersonPrx
PersonFactoryI::create(const string& name,
                          const Current& c)
{
    RWRecMutex::WLock lock(_lcMutex);// Write lock
    // ...
}
```

```
PersonSeq
PersonFactoryI::list(const Current& c) const
{
    RWRecMutex::RLock lock(_lcMutex); // Read lock
    // ...
}

PersonPrx
PersonFactoryI::find(const string& name,
                        const Current& c) const
{
    RWRecMutex::RLock lock(_lcMutex); // Read lock
    // ...
}
```

Note that this change also requires us to drop reaping in `list` and to only reap in `create` because reaping modifies the factory's state, but `list` acquires a read lock, not a write lock.

Similarly, you can use a read–write mutex for normal servant operations. For example, `getAddress` can acquire a read lock and `setAddress` can acquire a write lock, so concurrent calls to `getAddress` can proceed in parallel.

Be aware that providing such increased concurrency is worth it only if clients indeed call operations concurrently and, moreover, the operations are long-running. If either of these conditions does not hold, you are better off just sticking with the simple solution I presented earlier, namely, to strictly serialize operations on the factory and each servant. (It is easy to fall into the trap of premature optimization, especially with threaded programs. I urge you not to optimize unless you have performed an analysis to show that the optimization will actually optimize something.)

What's in a Name?

Earlier, I mentioned that using a person's name as the object identity might be a bad idea. Why is this? To understand the issue, consider the following scenario:

- Client A creates a new person with name Fred.
- Client A passes the proxy for the Fred person as a parameter of a remote call to another part of the system, say, server B.
- Server B remembers Fred's proxy.
- Client A decides that person Fred is no longer needed and invokes Fred's `destroy` operation.
- Some time later, client C creates a new person object whose name also happens to be Fred.
- Server B decides that it needs to get the address of the Fred person it was originally passed by client A and invokes the `getAddress` operation on its remembered proxy.

At this point, things are likely to go horribly wrong: server B thinks that it is obtaining the address of the original Fred person. However, that Fred person no longer exists: the Fred person that does exist is a completely different person (presumably with a completely different address).

What is happening here is that Fred has been reincarnated because the same object identity was used for two different Ice objects. While such object reincarnation might have a certain Hinduist appeal, it is generally a bad thing. In particular, consider the following two interfaces:

```
// Slice
interface Process
{
    void launch(); // Start process
    // ...
};

interface NuclearBomb
{
    void launch(); // Kill a lot of people
    // ...
};
```

Replaying the above scenario, if client A creates a `Process` object called "Little Boy" and destroys that object again, and client C creates a `NuclearBomb` called "Little Boy", when server B calls `launch`, it will launch a nuclear bomb instead of a process.[2] Admittedly, this example is contrived, but it illustrates an important point: when the Ice run time dispatches a request, exactly three items determine where the request ends up being processed:

- the endpoint at which the server listens for incoming requests
- the identity of the Ice object that is the target of the request
- the name of the operation that is to be invoked on the Ice object

It follows that, if object identities are insufficiently unique, it is possible for a request to be processed by an entirely unexpected Ice object, provided that object supports an operation with the same name, and that the parameters passed to one operation happen to decode correctly when interpreted as the parameters to the other operation. (This is rare, but not impossible, depending on the type and number of parameters.)

The crucial question is, what do we mean by "insufficiently unique"? As far as call dispatch is concerned, identities must be unique only per object adapter. This is because the ASM does not allow you to add two entries with the same object identity; by enforcing this, the ASM ensures that each object identity maps to exactly one servant. (Note that the converse, namely, that servants in ASM entries must be unique, is *not* the case: the ASM allows you to map different object identities to the same servant, which is useful if you want to implement stateless façade objects—see the Ice manual for more information on façade servants.) So, as far as the run time is concerned, it is perfectly OK for you to reuse object identities for different Ice objects.

2. This tacitly assumes that `Process` and `NuclearBomb` are implemented by the same object adapter, so requests for either interface end up being sent to the same endpoint.

So, why does the run time not prevent you from reusing identities? After all, that would avoid the problem entirely. The answer is that doing so would require the run time to remember every object identity that has ever been used—clearly, that is impossible because the run time would have to remember an unbounded amount of state. Ice deals with the problem by declaring it an SEP (Somebody Else's Problem).[3] In this particular case, that somebody else is you, the application developer. The Ice object model simply assumes that all object identities are globally unique in space and time, but it cannot enforce that assumption; the run time relies on you, the developer, to make object identities "sufficiently unique".

So, how should you deal with the problem? One option is to simply ignore it and do nothing. This sounds facetious, but isn't. For many applications, due to the way they are structured, the problem never arises: application-specific constraints automatically ensure that object identities are never reused. For example, if you use a social security number as a person's identity, the problem cannot arise because the social security number of a deceased person is not given to another person (or so one would hope…).

Another option is to allow identity reuse and to write your application components such that they can deal with reused object identities: if nothing bad happens when an identity is reused, there is no problem. (This will be the case if you know that the life cycles of two different objects with the same identity can never overlap.)

However, even if identity reuse does not pose a problem for your application, you should still give consideration to using globally unique identities for at least some of your objects. One reason for this is IceGrid: IceGrid supports *well-known proxies*. In stringified form, a well-known proxy looks like this:

```
Fred
```

Note that this proxy does not contain an endpoint or adapter name. The only thing it contains is an object identity. When a client uses such a proxy to invoke an operation, the Ice run time consults IceGrid behind the scenes to ask it where the server for this object can be found. In other words, servers can advertise the identities of specific objects with IceGrid and hand out proxies that only contain an object identity to clients. IceGrid maps well-known proxies to proxies with endpoints and returns a direct proxy (which contains the server's endpoints) to the client-side run time when a client invokes on a well-known proxy. The advantage of using well-known proxies is that they simplify client configuration. Also, because well-known (and indirect) proxies do not contain endpoint information, you can move a server to a different machine without invalidating the proxies that were given to clients.

3. With apologies to Douglas Adams, who created the SEP in *Life, the Universe, and Everything*.

For well-known proxies to work, they must be unique within an IceGrid domain, that is, none of the well-known objects of *all* servers that use a particular IceGrid registry can have the same identity. Of course, this requirement is a lot more stringent than uniqueness only within a single object adapter within a single server.

If you intend to use well-known proxies, your object identities must be unique within the IceGrid domain. Similarly, if you intend to support object migration (that is, moving a number of object implementations from one server to another to rebalance load), object identities must be unique across the source and target server. So, at least for your well-known objects (typically, a handful of key objects that clients need for bootstrapping), you should consider using globally unique identities. Fortunately, Ice makes this easy. You can generate an object identity that is a universally unique identifier (UUID) by calling `addWithUUID` on the object adapter, instead of calling `add` and supplying your own object identity

UUIDs are guaranteed to be unique forever,[4] so it becomes impossible to intentionally or accidentally reuse an object identity.

Now, before you go and rush off and change all your object identities to UUIDs, keep in mind that they do add a little bit of overhead. For one, UUIDs occupy 36 bytes, so they can be larger than an Ice object's "natural" identity. Second, if you want to store object state in a database, you must add an extra field to your database table to store the UUID. However, this is rarely an issue, unless objects contain only a very small amount of state. (You might also want to check out Matthew Newhook's article *A Persistent Chat Server* in Issue 7 of *Connections*, which also discusses UUIDs and persistence.)

In general, I recommend that, if an Ice object naturally contains a unique item of state (such as a social security number), you should use that item as the object identity. On the other hand, if the natural object identity is insufficiently unique, you can use a UUID as the object identity. (This is particularly useful for anonymous transient objects, such as session objects, that may not have a natural identity.)

For well-known objects, you can either use a UUID, or establish uniqueness by convention. For example, you could use a domain name and department prefix (or any other suitable administrative convention) to create identities for well-known objects, such as `com.AcmeCorp.ProjectX.Testing.Fred`. This has the advantage of providing human-readable unique identities, which makes configuration a little easier. (UUIDs look something like 746ccfc0-7907-4002-92df-0be5b72b4abb and are therefore not all that user-friendly.)

4. Well, depending on the implementation, not quite guaranteed, but generating the same UUID twice is so unlikely that there is no point in worrying about it.

Recap

Having read through the preceding text, you may be surprised at how difficult it is to provide life cycle operations for such a simple application. Supporting `create` is usually straightforward and does not add much complexity, but supporting `destroy` requires careful thought because the actions of `destroy` can interfere with the semantics of other operations; if we want to provide concurrent access for clients, the implementation requires some care. (Incidentally, this is a common theme in distributed computing: tearing things down cleanly is usually a *lot* harder than setting them up; most programmers are quite spoiled by the ability to simply call `exit` and have the operating system worry about how to clean up the mess.) In general, you should give the reaping approach serious consideration: it is less likely to cause deadlock, easier to understand, avoids circular dependencies, and applies more generally than the callback approach.

I hope you will find this article useful the next time you contemplate the wheel of life and death of your objects. As usual, you can download the source code for this article from our web site. Feel free to use the code as is or to modify it to suit your requirements. Please contact me in ZeroC's forum if you would like to further discuss object life cycle.

FAQ Corner

In each issue of our newsletter, we present a few frequently-asked questions about Ice. The questions and answers are taken from our support forum at http://www.zeroc.com/vbulletin/ and deal with specific problems that developers tend to encounter, and for which the answer may not be readily apparent from reading the documentation. We hope that you will find the hints and explanations in this section useful.

Q: How do I use ACE and Ice together?

When you try to use ACE and Ice in the same C++ program, you might encounter a compilation error similar to this one:

```
C:\Ice\include\Ice/Application.h(33) : error
C2487: 'ace_os_main_i' : member of dll interface
class may not be declared with dll interface
```

The problem is caused by a macro that ACE uses to redefine `main`:

```
#     define main \
ace_os_main_i (int, char *[]); //...
```

As you might expect, this macro causes conflicts with member functions named `main`, such as `Ice::Application::main`.

You can work around this problem by rearranging your header files: first include all necessary Ice header files, then include the ACE header files.

Q: How can I increase the maximum number of threads my C++ application can create?

A program that creates many threads is generally restrained by the availability of virtual memory because the operating system allocates a stack for each new thread. The default size of this stack varies by operating system; on Windows and Linux, the default size is 1MB. This value is typically sufficient for most needs, but we clearly cannot expect it to be appropriate for every application. If a 32-bit program is creating hundreds or thousands of threads, it's going to hit the virtual memory limit quite soon if it accepts the default stack size.

The thread's stack is consumed by local variables, nested function calls, and recursion, therefore selecting a smaller stack size should only be done with careful consideration (and a lot of testing!). You, as the developer, are the best judge of a suitable size for the stack.

If you decide to change the stack size, you have several options. For example, many linkers allow you to specify a different default stack size, which affects every thread the program creates. You can also change the size more selectively by specifying it individually for each thread. The `IceUtil::Thread` class allows you to supply a different size for the stack when calling the `start` method, as shown below:

```
IceUtil::ThreadPtr t = ...
t->start(32 * 1024); // 32K stack
```

In Ice applications, the Ice run time uses configuration properties to determine the stack size of the threads it creates. For example, the default client and server thread pools use the properties `Ice.ThreadPool.Client.StackSize` and `Ice.ThreadPool.Server.StackSize`, respectively. If you've configured an object adapter `MyAdapter` with its own thread pool, the stack size configuration property is named `MyAdapter.ThreadPool.StackSize`.

When using the thread-per-connection concurrency model, in which the Ice run time creates a new thread for each connection, the relevant configuration property is `Ice.ThreadPerConnection.StackSize`. This property can be especially important when configuring a Glacier2 router that handles a large number of connections.

Finally, it's important to understand that it is rarely advantageous to create lots of threads. For compute-bound applications, it's best to limit the number of threads to the number of physical processors in the host machine; adding any more threads only increases context switches and reduces performance. Additional threads can improve responsiveness when threads can become blocked while waiting for the operating system to complete a task, such as a network or file operation. However, having an abundance of threads is not like having an abundance of beer: if performance is good with ten threads, it will not be as good with one thousand threads, as all of those threads are competing for processor time and will drag your system's performance to its knees.