Connections

ZeroC's Newsletter for the Ice Community

Issue Number 3, June 2005

Invent It or Use It?

I recently had a conversation with a prospective customer about an application development project. The application required publish–subscribe functionality, but had no other unusual distribution requirements. I was amazed when I learned that the customer was seriously considering implementing a complete publish–subscribe middleware layer for this application, even though middleware was neither the focus of the customer's business, nor did the customer have any in-depth experience in developing communications software. Unfortunately, I see this sort of thinking a lot: the "not-invented-here" syndrome is alive and well, often driven by the desire of engineers to work on what they find interesting, rather than build what is needed in order to make a profit.

Yes, building middleware can be rewarding and a lot of fun. At least for a while—until reality sets in and you end up spending weeks of effort porting to new platforms, working around bugs in compilers and TCP/IP implementations, or hunting down that pesky race condition that shows up only once every two weeks. And you spend even more time improving performance, reducing bandwidth consumption, creating portability libraries, and inventing error handling strategies. I could go on in this vein for a long time—suffice it to say that many thousands of hours of work by very experienced developers have gone into Ice in order to turn it into a robust, reliable, portable, and high-performance product.

The chance that any in-house middleware development, especially when carried out by people with little experience in the field, will end up being cost-effective is very slim indeed. And even slimmer is the chance that the result will end up performing significantly better than Ice. For our customer, after a closer look at the requirements, IceStorm turned out to be a good solution, and one that did not require inventing anything.

Which brings me to Matthew Newhook's article in this issue… Matthew shows you how you can use IceStorm for your own applications, and with surprisingly little effort. Please, give this a good look—chances are that you will consider it time well spent. And, trust me: developing your own middleware is a lot less rewarding and fun than you might think!

Michi Henning
Chief Scientist
ZeroC, Inc.

Issue Features

An Introduction to IceStorm

In this installment of his multi-part article series, Matthew Newhook shows you how to extend the chat room application to support multiple chat rooms with the help of IceStorm.

Interpreted Ice

Ice and Python form a powerful combination for rapid prototyping, debugging, and maintenance. This article provides many tips and tricks that you can use to make your development more productive, regardless of whether your application is written in Python or another language.

The Grim Reaper

Stateful interactions between client and server are a fact of life and raise the problem of how to clean up server-side state in case something goes wrong with the client. This article presents a simple and effective way to get rid of stale objects in a server without writing reams of code.

Contents

An Introduction to IceStorm

Matthew Newhook, Senior Software Engineer

Introduction

The previous article focused on using advanced features of Glacier2 to make the chat server more robust. This article discusses how to add support for multiple chat rooms to the chat server using IceStorm.

Interface Changes

The chat server we have developed up to this point is very limited. An obvious extension to its functionality is to add support for multiple chat rooms. First a review of the current `ChatSession` interface:

```
// Slice
interface ChatCallback
{
    void message(string data);
};
interface ChatSession
    extends Glacier2::Session
{
    void setCallback(ChatCallback* callback);
    void say(string data);
};
```

From the perspective of a client, a `ChatSession` object has two functions: it allows the client to send messages to a room, and it informs the client when a message has arrived from a room via the callback object. To extend this functionality to multiple rooms, we could identify each chat room by a name and pass the room name to the `say` and `message` operations, as follows:

```
// Slice
interface ChatCallback
{
    void message(string room, string data);
};
interface ChatSession
    extends Glacier2::Session
{
    void setCallback(ChatCallback* callback);
    void say(string room, string data);
};
```

This approach, while viable, isn't very clean and is definitely not object-oriented. It also lacks extensibility: as our application evolves, we expect to add other room-specific data, such as a chat room participant list.

A better approach is to use a separate interface that receives state changes, but to make the identity of chat room implicit in the instance of the interface:

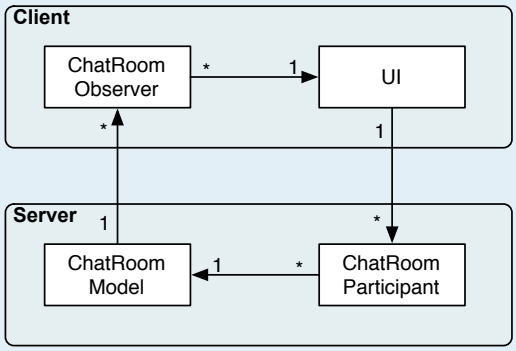
```
// Slice
interface ChatRoomObserver
{
    void message(string data);
};
```

This interface is identical to the `ChatRoomCallback` class in the previous implementation but, by using separate instances of this interface, one for each chatroom, the client can receive messages from different rooms. Similarly, we need an interface that allows the client to send a message to a specific room; again, we can do this with the room implicitly identified by the instance:

```
// Slice
interface ChatRoomParticipant
{
    void say(string data);
};
```

See Figure 1 for a class diagram. (The `ChatRoomModel` object in the diagram is purely a server side-concept, and is not accessible to the client.)

Figure 1: Class Diagram



We also need a method to allow the client to join a chat room. Where should this go? Access to all the server-side functionality is provided by the session interface, so this seems like a logical place:

```
// Slice
interface ChatSession
    extends Glacier2::Session
{
    ChatRoomParticipant* join(string room);
};
```

This allows the user to join a chat room. However, the new interfaces require one `ChatRoomObserver` proxy per chat room, not a single `ChatRoomObserver` for all chat rooms. The simplest way to deal with this is to pass the `ChatRoomObserver` proxy as an extra parameter to the `join` operation. We can remove the `setCallback` operation that was formerly on `ChatSession`, as well as the `say` operation (which has been replaced by `ChatRoomParticipant::say`). This leaves the following:

```
// Slice
interface ChatSession
    extends Glacier2::Session
{
    ChatRoomParticipant* join(string room,
        ChatRoomObserver* observer);
};
```

Client-Side Support

Let's run through the changes that are required in the client to support multiple chat rooms.

Firstly, we replace the `ChatCallback` implementation with the `ChatRoomObserver` implementation. It is largely unchanged from its previous version:

```
// C++
class ChatRoomObserverI : public ChatRoomObserver
{
public:
    ChatRoomObserverI(const string& room) :
        _room(room)
    {
    }
    virtual void message(const string& data,
        const Current&)
    {
        cout << _room << ": " << data << endl;
    }

private:
    const string _room;
};
```

Next, we need to have a way of keeping track of the chat rooms the client has joined. We'll add a new class called `ChatRoomManager`:

```
// C++
class ChatRoomManager
{
public:
    ChatRoomParticipantPrx find(
        const string& room) const;
    void join(const string& roomName);
};
```

The class maintains a map of room names to `ChatRoomParticipant` proxies.

```
map<string, ChatRoomParticipantPrx> _rooms;
```

The implementation of `join` then is as follows:

```
// C++
void ChatRoomManager::join(const string& roomName)
{
    Identity id;
    id.name = "observer." + roomName;
    id.category = _category;
    ChatRoomObserverPrx observer =
        ChatRoomObserverPrx::uncheckedCast(
            _adapter->add(
            new ChatRoomObserverI(roomName),id));
    ChatRoomParticipantPrx room = _session->join(
        roomName, observer);
    _rooms.insert(make_pair(roomName, room));
}
```

The object identity name is constructed from the name of the room. (Recall that Glacier2 requires all client-side callback objects to use a specific per-session category, as described in *Session Management with Glacier2* in Issue 1 of *Connections*.) The remainder of the client-side implementation is straightforward—see the source code for this application in the Ice distribution.

Server-Side Support

The server-side needs to be modified to support the new interfaces as well.

ChatSessionI Implementation

First, we'll look at the implementation of `ChatSessionI`.

```
// C++
ChatRoomParticipantPrx ChatSessionI::join(
    const string& room,
    const ChatRoomObserverPrx& observer,
    const Current& current)
{
    Lock sync(*this);
    Identity id;
    id.category = "_" + _userId;
    id.name = IceUtil::generateUUID();
    ChatRoomParticipantPrx proxy =
        ChatRoomParticipantPrx::uncheckedCast(
        current.adapter->add(
            new ChatRoomParticipantI(
                room, _userId, observer), id));
    _rooms.push_back(proxy);
    return proxy;
}
```

```
void ChatSessionI::destroy(const Current& current)
{
    Lock sync(*this);
    current.adapter->remove(current.id);
    list<ChatRoomParticipantPrx>::
        const_iterator p;
    for(p = _rooms.begin();
        p != _rooms.end();
        ++p)
    {
        current.adapter->remove(
            (*p)->ice_getIdentity());
    }
    _rooms.clear();
}
```

`ChatSessionI` maintains a list of `ChatRoomParticipantPrx` that the user has created during a session. `ChatSessionI::join` creates a new `ChatRoomParticipant` Ice object, adds the proxy to the list of rooms, and returns the proxy to the caller. Note that the category of the `ChatRoomParticipant` Ice object must be `"_" + userId`. This is because of the object-level restrictions that we have asked Glacier2 to impose. (See *Advanced Use of Glacier2* in Issue 2 of *Connections*).

`ChatSessionI::destroy` runs through the list of `ChatRoomParticipant` proxies, removes each object from the object adapter, and then destroys the session itself. (Removing the objects from the adapter is necessary to avoid a memory leak.)

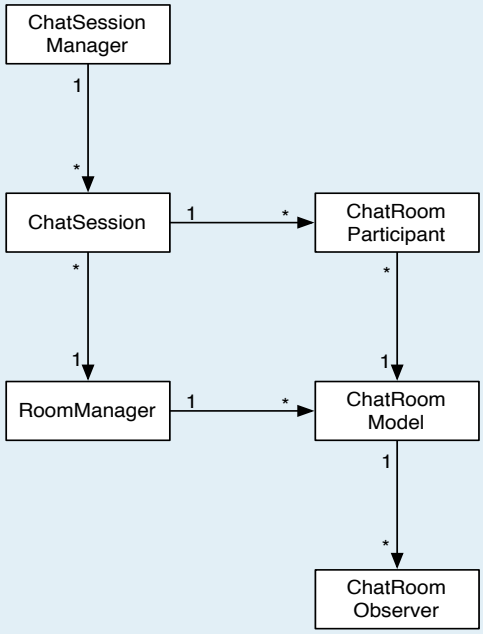
Multiple Chat Rooms

The chat server to this point only supports one chat room. The chat room model was formerly implemented by a class called `ChatRoom`. For consistency with the new class names that we are using for this version of the application, we'll rename this to `ChatRoomModel`. The definition is as follows:

```
// C++
class ChatRoomModel : public IceUtil::Mutex,
                      public IceUtil::Shared
{
public:
    void enter(const ChatRoomObserverPrx&);
    void leave(const ChatRoomObserverPrx&);
    void message(const string&);
};
```

One approach to support multiple chat rooms is to add a class `RoomManager` that maintains a list of `ChatRoomModel` instances, one per chat room. To make the chat server scale, it must be possible to distribute these chat rooms over a series of servers. It follows that `ChatRoomModel` must be an Ice object. Figure 2 shows a class diagram of this architecture.

Figure 2: Proposed Class Diagram



Before we go too far down this road, let's take a step back for a moment. What does the `ChatRoomModel` class really do? It maintains a list of connected subscribers and distributes events to these subscribers. Let's take a look at an example of the event distribution code as it currently stands:

```
// C++
void ChatRoomModel::message(
    const string& data) const
{
    Lock sync(*this);
    list<ChatCallbackPrx>::const_iterator p;
    for(p = _members.begin();
        p != _members.end();
        ++p)
    {
        try
        {
            (*p)->message(data);
        }
        catch(const LocalException&)
        {
        }
    }
}
```

Now, consider: what would happen if were to add new types of events `ChatRoomObserver` interface, such as an event to indicate that a chat room has been closed? The resulting change would be as follows:

```
// Slice
interface ChatRoomObserver
{
    void message(string data);
    void destroy();
};
```

In turn, this would force us to add another method to `ChatRoomModel` to send the destroy event:

```
// C++
void ChatRoomModel::destroy(
    const string& data) const
{
    Lock sync(*this);
    list<ChatCallbackPrx>::const_iterator p;
    for(p = _members.begin();
        p != _members.end();
        ++p)
    {
        try
        {
            (*p)->destroy();
        }
        catch(const LocalException&)
        {
        }
    }
}
```

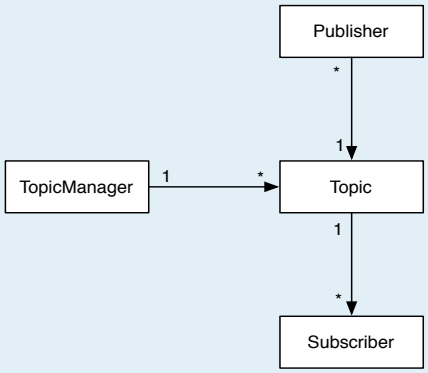
There may well be other events we would like to add to `ChatRoomObserver`, such as notification of a member entering or exiting a room. Each of these new events would require us to write new methods to distribute the event to the chat room observers—surely there is a better solution? The answer is, fortunately, yes. Ice has the perfect solution to this problem: IceStorm.

IceStorm

IceStorm is a publish–subscribe service. Publishers send events to connected subscribers. IceStorm manages a set of event channels called topics. Each topic is associated with a user-defined Slice interface or class, all of whose operations must be callable as oneway: they must have `void` return type, have no out-parameters, and cannot throw user exceptions. Publishers use a topic to send events to subscribers by calling operations on a proxy for the topic. Figure 3 shows an object diagram for this.

The `TopicManager` allows applications to create and retrieve `Topic` objects. (The interface definitions that follow are incomplete and leave out portions that are not relevant to this article. For the complete documented interfaces please see the Ice manual.) The salient pieces of the interface are as follows.

Figure 3: IceStorm Diagram




```
// Slice
interface TopicManager
{
    Topic* create(string name)
        throws TopicExists;
    nonmutating Topic* retrieve(string name)
        throws NoSuchTopic;
};
```

A `Topic` allows the connection and disconnection of subscribers as well as the publishing of events. The abbreviated interface follows:

```
// Slice
interface Topic
{
    nonmutating Object* getPublisher();
    void subscribe(QoS theQoS,
        Object* subscriber);
    idempotent void unsubscribe(
        Object* subscriber);
    void destroy();
};
```

Using IceStorm in an application is very simple. The basic steps are as follows:

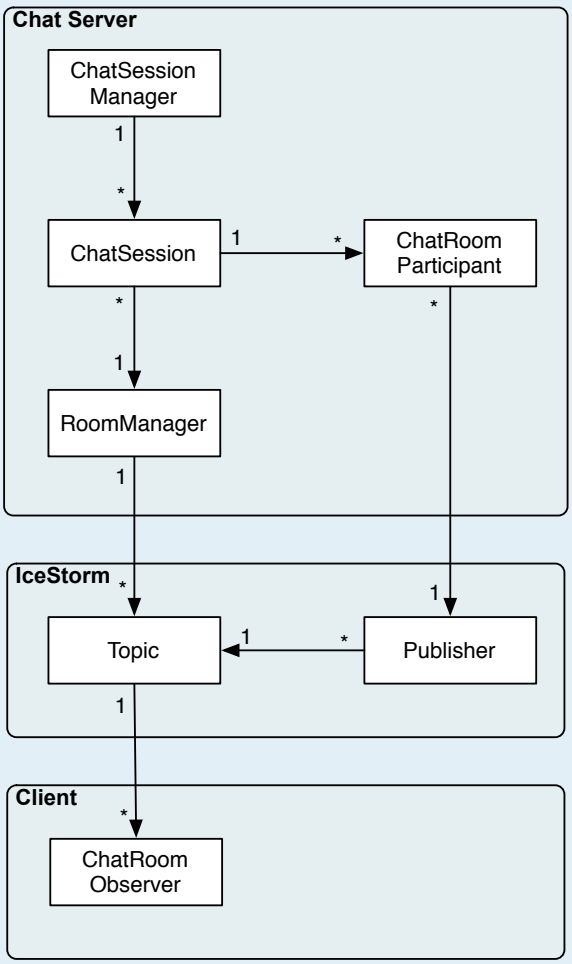
1. Create an interface to describe the event data being distributed. In our case we already have this interface, namely `ChatRoomObserver`.
2. Create a topic. Events are distributed on a topic to all connected subscribers.
3. Subscribers must implement an interface that IceStorm calls to deliver events. We've already done this on the client side by implementing the `ChatRoomObserver` interface.
4. Connect any interested subscribers to the topic.
5. Publishers publish events on the topic. This is accomplished by getting a publishing proxy, casting it to the correct type, and then calling methods on the proxy.

The chat server can use IceStorm to send chat events to `ChatRoomObserver` instances. As opposed to the `ChatRoom` solution that we discussed above, we don't have to write any code to

manage the subscriber set or to distribute events to the connected subscribers. No new code has to be written if new methods are added to the `ChatRoomObserver` interface and, if we encounter a scalability problem in the future, we can deal with it by deploying more IceStorm servers.

Each chat room is represented by a separate topic. The name of the topic is the name of the chat room. In essence, the IceStorm topic implements the `ChatRoomModel`. Figure 4 shows the architecture of the system.

Figure 4: Architecture with IceStorm



RoomManager

The first step is to write some code to manage the mapping of chat room name to topics. We'll encapsulate this implementation in a class named `RoomManager`.

As was the case for the `ChatRoom` class, we'll need methods to add and remove subscribers to a particular chat room. The interface therefore looks something like this:

```
// C++
class RoomManager
{
public:
    void enter(const string& room,
        const ChatRoomObserverPrx& observer);
    void leave(const string& room,
        const ChatRoomObserverPrx& observer);
};
```

The steps required to add a subscriber to the chat room are:

1. Create the topic, if necessary.
2. Subscribe the observer. Because we leave the quality-of-service parameter (`IceStorm::QoS`) empty, IceStorm uses its configured defaults. (See the Ice manual for more information on IceStorm's quality-of-service parameters.)

The implementation is as follows. (As usual, for clarity, we omit error handling.) `_manager` is a proxy to the `IceStorm::TopicManager` interface.

```
// C++
void RoomManager::enter(const string& room,
    const ChatRoomObserverPrx& observer)
{
    Lock sync(*this);
    TopicPrx topic;
    map<string, IceStorm::TopicPrx>::
    const_iterator p = _rooms.find(room);
    if(p == _rooms.end())
    {
        topic = _manager->create(room);
        _rooms.insert(make_pair(room, topic));
    }
    else
    {
        topic = p->second;
    }
    topic->subscribe(IceStorm::QoS(), observer);
}
```

The implementation maintains a dictionary of room name to `IceStorm::TopicPrx`. This avoids having to make a remote method invocation on the IceStorm `TopicManager` to locate the topic proxy for each `RoomManager::enter`, and `RoomManager::leave` call.

`RoomManager::leave` is very simple: it locates the topic in the dictionary and then unsubscribe the observer:

```
// C++
void RoomManager::leave(const string& room,
    const ChatRoomObserverPrx& observer)
{
    Lock sync(*this);
    map<string, IceStorm::TopicPrx>::iterator
        p = _rooms.find(room);
    p->second->unsubscribe(observer);
}
```

All that remains is to hook up this implementation with the `ChatRoomParticipant` interface. (Recall that this is the interface that is called by the client to publish a message on a chat room.)

```
// C++
class ChatRoomParticipantI :
    public ChatRoomParticipant
{
public:
    ChatRoomParticipantI(
        const string& room,
        const string& userId,
        const ChatRoomObserverPrx& observer) :
        _room(room),
        _userId(userId),
        _observer(observer),
    {
        RoomManager::instance()->enter(
            room, observer);
    }
    ~ChatRoomParticipantI()
    {
        RoomManager::instance()->leave(
            _room, _observer);
    }
    virtual void say(const string& data,
        const Current&)
    {
        // ???
    }
};
```

There is clearly a problem. What does `say` do? `say` must publish an event to an IceStorm topic, by calling a method on the publisher proxy. But where does this proxy come from? IceStorm makes it available via the `IceStorm::Topic::getPublisher` operation. Since the `RoomManager` manages the topics, we can get the proxy from the manager we subscribe an observer. Therefore, we'll change the implementation of `RoomManager::enter` to return the topic publisher proxy, as follows:

```
// C++
IceStorm::TopicPrx topic = ...;
return ChatRoomObserverPrx::uncheckedCast(
    topic->getPublisher());
```

With this, the `ChatRoomParticipantI` constructor now stores the publisher proxy in a member variable `_publish`, and we can implement `ChatRoomParticipantI::say` in the obvious way:

```
// C++
virtual void say(const string& data,
    const Current&)
{
    _publish->message(_userId + " says: " + data);
}
```

Glacier2 Context Information

As discussed in *Advanced Use of Glacier2* in Issue 2 of *Connections*, we must pass information to a Glacier2 router in a `Context` to ensure that Glacier2 sends messages to the observer as oneway invocations with compression enabled. How can we pass this information using IceStorm? We could try to set the context on the proxy that is subscribed to the IceStorm topic:

```
// C++
Context context;
context["_fwd"] = "Oz";
IceStorm::TopicPrx topic = ...;
topic->subscribe(IceStorm::QoS(),
    observer->ice_newContext(context));
```

However, this does not work, because this context information is not part of the proxy, but is part of the information sent when we make a call on the proxy. The correct solution is to set the context information on the *publisher* proxy instead:

```
// C++
Context context;
context["_fwd"] = "Oz";
IceStorm::TopicPrx topic = ...;
return ChatRoomObserverPrx::uncheckedCast(
    topic->getPublisher()->
        ice_newContext(context));
```

Summary

Using IceStorm, we added multiple chat rooms to the server with very little code. (In fact, the server line count has actually decreased from the version in Issue 2!) In addition, it is now easier to add new features to the server, and we end up with a server that is both more flexible and more scalable than its previous incarnation.

Note that the chat client itself is completely unaware of IceStorm. Integrating IceStorm with the server backend did not change the client side interfaces at all, which is quite a remarkable feat!

If you have an application with any kind of publish–subscribe functionality, you should give IceStorm serious consideration:

1. You do not need to write any event distribution code, nor manage misbehaved subscribers.
2. You can leave your interfaces unchanged; IceStorm does not require any special "event data types."
3. Via federation, IceStorm scales extremely well. (See the Ice Manual for more information on federation.)
4. IceStorm various levels of quality of service, which allow you to easily fine-tune event delivery for the needs of your application without writing extra code.

Interpreted Ice: Distributed Application Development on Steroids

Brent Eagles, Senior Software Engineer

Introduction

During a project, we often find ourselves dealing with a large number of unknowns. While a lot of time can be spent writing design documents and attending meetings, at some point we have to sit down and write code. Often we will find that our assumptions that fed the design and planning process were inaccurate or just plain wrong. If the solution is to be distributed, it can greatly increase the amount of things that we can get wrong. While Ice takes much of the pain out of creating distributed applications, how best to design your architecture and distribute your workloads is still a bit of an art.

The sooner we can see how to design an architecture and identify the real problems and obstacles at hand, the sooner we can establish accurate schedules and deadlines, nail down requirements, and acquire resources. What we need is a toolset that allows us to quickly dive in and test our assumptions, so the overall shape of our applications can take form quickly. The toolset should also allow and encourage the creation of high-quality maintainable code so the system can evolve and we do not waste time refactoring poor-quality code.

Ice for Python *is* that toolset. Ice for Python brings all of the great things about Ice and Python together. What that means is that we have a dynamic and interpreted distributed programming environment that enables rapid code creation, testing, and modification. With this system, we can:

- quickly explore our assumptions and experiment with code,
- develop high-quality prototypes that can be evolved into the final product,
- Test and debug our system thoroughly without complicated third-party tools or debuggers.

Let's look at some of the features of the Ice + Python system and how they can affect how we develop distributed applications.

Programming Using an Interpreter

Code interpreters are everywhere. If you think that you have never used an interpreter, think again. If you have ever written or modified a shell script or batch file, then you have worked on a program that runs in an interpreter.

Being an interpreted language is among Python's many appealing features. While interpreters are often disparaged because of relatively slow performance, programming with them has advantages: interpreters reduce development time by simplifying or omitting the edit/compile/run cycle, and they often allow the developer to directly interact and modify code in the runtime environment; this facilitates rapid experimentation and testing of new code.

Running the Python Interpreter in Interactive Mode

To work with the Python interpreter in interactive mode, simply run Python without passing it a source file. You will be greeted by a short message and a prompt that might look something like the following:

```
Python 2.3.4 (#1, Feb 2 2005, 12:11:53)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on
linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

Let's start our interactive experience by building the traditional first program, the venerable *Hello World* application.

```
# Python Interpreter
>>> print 'Hello World'
```

Pressing enter yields:

```
Hello World
```

Programming without functions is boring, so let's try putting the *Hello World* "logic" into a function.

```
# Python Interpreter
>>> def HelloWorld():
    print 'Hello World'
>>>
```

If we want to test this function to make sure it works, we can try it right away.

```
>>> HelloWorld()
Hello World
```

Modifying Code on the Fly

So far we have created new code in our interpreter, but what do we do if we want to change what the `HelloWorld` function does? Do we have to restart the interpreter and start over? No, we can redefine the function immediately in the interpreter. Our first *Hello World* function is pretty dull because it does not have enough emphasis! We want to greet the world with lots of enthusiasm, so let's modify our output a little.

```
# Python Interpreter
>>> def HelloWorld():
    print 'Hello World!'

>>> HelloWorld()
HelloWorld!
```

This is pretty economical development: we wrote a *Hello World* program in Python, modified it twice, ran all versions without leav-

ing the Python environment, and we hit that enter key fewer than 10 times!

Being able to easily create, modify, and restructure code on the fly is useful during the start-up phase of a project. But writing a large program directly in the interpreter would quickly become cumbersome, and it would not take long for us to miss the functionality of our favorite editor. Fortunately, we can use an editor to write the bulk of the code, import it into a running Python environment, and then run it interactively. Let's put our function in a file called `Greetings.py`:

```
# Greetings.py
def HelloWorld():
    print 'Hello World'
```

Now we can import the module into a running Python session and call our greeting function:

```
# Python Interpreter
>>> from Greetings import *
>>> HelloWorld()
Hello World
```

The line `from Greetings import *` imports all of the functions from the module `Greetings` into the local namespace. We could simply write

```
# Python Interpreter
>>> import Greetings
```

and then call the `HelloWorld` function through its fully qualified name, including the `Greetings` namespace:

```
# Python Interpreter
>>> Greetings.HelloWorld()
Hello World
```

If we need to modify the `HelloWorld` function now, we can do it interactively as before, or we can edit the source file and reload it. Here is the modified `Greetings.py`:

```
def HelloWorld()
    print 'Hello World!'
```

And here is how to reload the file into the interpreter:

```
>>>
>>> reload(Greetings)
>>> HelloWorld()
Hello World!
```

While this is a trivial demonstration of interactive programming, the implications are intriguing and exciting. Imagine

- being able to write high-level routines that interact with servers and make ad-hoc queries and tests on the fly,
- writing a server that runs in an interactive session, providing immediate feedback on server status and allowing control of the server operation without the overhead of developing custom user interfaces,
- dynamically modifying behavior by modifying object and function definitions at runtime.

We could even transfer Python code over the wire and run it on the server! While there are obvious safety and security issues with this capability, it is a powerful feature for implementing administrative functions.

Debugging

Debugging in an interactive interpreter session is very similar to debugging compiled programs. A debugger is essentially a mechanism for interactively stepping through code and examining state, much like an interactive interpreter does. The difference is that, in an interpreted system, each line of code is parsed and executed a line at a time, while, in a compiled program, the executing machine code is mapped to source through debug information. Besides stepping through code, some debuggers permit you to modify values during execution. This is a valuable feature for synthesizing error conditions and debugging hard-to-reach code. While most good debuggers allow you to do this with compiled code, debugging with interpreters has the extra advantage that we have more control over execution. We can:

- modify code at runtime

  This allows us to fix bugs as soon as we see them and possibly continue execution of the program.
- call arbitrary functions

  We can call functions to test the state of the program at any time and synthesize hypothetical situations. This includes creating new objects, files, etc.

The Python standard library contains a module called `inspect` that is valuable for (you guessed it) inspecting objects in the Python environment.

The Python interpreter, Ice, and You

OK, so we have determined that running an interpreter in interactive mode has some possibilities. How does this affect how we work with Ice? Let's look at some possibilities. As an example, we use a simple sorting server.

```
// Slice Utility.ice
module Utility
{
    typedef sequence<int> IntegerSequence;
    interface Sorter
    {
        // Returns a sorted copy of the provided
        // integer sequence
        IntegerSequence sortIntegers(
            IntegerSequence unsorted);
    };
};
```

```
# Python UtilityServer.py
import sys, traceback, Ice
```

```
Ice.loadSlice('Utility.ice')
import Utility

class SorterI(Utility.Sorter):
    def sortIntegers(self, unsorted,\
            current = None):
        sorted = unsorted
        j = 0
        while j < len(sorted) - 1:
            i = 0
            while i < len(sorted) - 1 - j:
            if sorted[i + 1] < sorted[i]:
                t = sorted[i]
                sorted[i] = sorted[i + 1]
                sorted[i+ 1] = t
            i = i + 1
            j = j + 1
        return sorted

class SortServer(Ice.Application):
    def run(self, args):
        adapter = self.communicator().\
            createObjectAdapter('Utility')
        object = SorterI()
        adapter.add(\
            object,Ice.stringToIdentity('sorter'))
        adapter.activate()
        communicator.waitForShutdown()
        return 0

app = SortServer()
sys.exit(app.main(sys.argv))
```

Load It Up with Ice.loadSlice()

Ice for Python takes full advantage of the interpreted environment by providing the ability to translate and compile Slice code using the `Ice.loadSlice()` command. No more makefiles! Making the translation step part of the initialization process for the code and skipping a build step simplifies adding, modifying, and removing Slice. The improved agility is a boon during the exploratory and experimentation phases of a project. With fewer steps involved in changing and using Slice, you are less likely to avoid necessary changes, and you have more time to spend constructing good tests.

Structuring Code for Flexibility

We have some server code and we want to test our sorting routine. Unfortunately, we find that the structure of our server code limits us to simply running it. This is because the initialization of the application is in the main line of execution. A better strategy is to start the server code only if the code is being run as a script. We can do this by checking the `__name__` property, as follows:

```
# Python
if __name__ == '__main__':
    app = SortServer()
    sys.exit(app.main(sys.argv))
```

If we run the code as a script (e.g. `python UtilityServer.py`), it runs as before, but now we can also import the code into a test suite or interactive session and test it.

```
# Python Interpreter
>>> import UtilityServer
>>> sorter = UtilityServer.SorterI()
>>> print sorter.sortIntegers([5, 4, 10, 3])
[3, 4, 5, 10]
```

Satisfied that the basics are working, we can now proceed with constructing a more exhaustive unit test by copying what we've written and adding more test data sets.

What if we found a bug when we called `SorterI.sortIntegers()`? One approach would be to exit the interpreter and start over, but there is another way. We can reload the module using the `reload()` function, create a new instance of the object, and try again.

```
# Python Interpreter
>>> reload(UtilityServer)
<module 'UtilityServer' from 'UtilityServer.pyc'>
>>> sorter = UtilityServer.SorterI()
>>> print sorter.sortIntegers([5, 4, 10, 3])
[3, 4, 5, 10]
```

We need to create a new instance of the sorter object because reloading code doesn't affect previously instantiated objects. If there is a situation where creating new instances of objects is problematic and we want to be able to reload code, we can restructure our code even further by delegating to helper methods. Moving our sort implementation into a helper method and modifying our servant, our server code becomes:

```
# Python
def sortImpl(unsorted):
    values = []
    values.extend(unsorted)
    j = 0
    while j < len(values) - 1:
        i = 0
        while i < len(values) - 1 - j:
            if values[i + 1] < values[i]:
                t = values[i]
                values[i] = values[i + 1]
                values[i + 1] = t
            i = i + 1
        j = j + 1
    return values

class SorterI(Utility.Sorter):
    def sortIntegers(self, unsorted,\
            current = None):
        return sortImpl(unsorted)
```

We can now easily change our sorting algorithm to improve efficiency, debugging, etc. At some point, however, we will probably realize that we've been wasting our time since Python implements a sort routine for us! We could try it right away by modifying our `sortImpl` implementation to delegate to it:

```
def sortImpl(unsorted):
    values = []
    values.extend(unsorted)
    values.sort()
    return values
```

In most situations, it is probably overkill to implement objects in terms of helper methods, just so we can change behaviors and fix bugs at runtime. However, it is useful when experimenting with code or implementing critical code that might need updating while a program is running.

In contrast, restructuring source code so it can be used as a module as well as a script is useful because it facilitates testing and code reuse. If a Python source file is meant to be primarily used as a module, we can use the same trick in reverse to create self-testing modules. For example:

```
if __name__ == '__main__':
    runTests()
```

To run the tests, simply use the file as a script:

```
python myFunctionLibrary.py
```

If the file is imported as a module, the tests will not be run.

Interacting with Remote Objects

We've seen some of the advantages of changing our approach to implementing servers to exploit the Python interpreter's interactive mode. The usefulness of Python's interactive mode is not limited to implementing servers. A little client boiler plate allows us to interactively make calls on remote objects. For example:

```
# Python Interpreter
>>> import sys, traceback, Ice
>>> Ice.loadSlice('Utility.ice')
>>> import Utility
>>> communicator = Ice.initialize([])
>>> obj = communicator.stringToProxy(\
    'sorter:tcp -p 15210')
>>> sorter = Utility.SorterPrx.checkedCast(obj)
>>> print sorter
sorter -t:tcp -h 192.168.1.6 -p 15210
>>> print sorter.sortIntegers(\
    [45, 32, 1, 56, 102])
[1, 32, 45, 56, 102]
>>> print sorter.sortIntegers([3])
[3]
>>> print sorter.sortIntegers([])
[]
```

How can this help us when working with distributed systems? Here are some examples:

Dealing with Black Boxes

It is not uncommon to use objects that were implemented by other developers. In such a situation, we often do not have access to source code or to other resources required to unit test or otherwise verify the correctness of the implementation, but we still want to implement an acceptance test suite. We can use a Python session to interactively access remote objects to try out scenarios that should be added to the acceptance test suite. If we find a problem, we can send snippets of Python code back to the third-party developers so they can test on-site, or we can get on the phone and collaboratively test a server running in a debugger. Imagine a dialog like the following:

> Sanjay: We've discovered what we think is a problem in your implementation of `Utility::Sorter::sortIntegers()`.
>
> Bob: I see. I happen to have this code running in a debugger right now. You should be able to access it at xyz.foo.com on port 8721. Give me a call when you are ready to test and what sequence of values you are going to send.
>
> Sanjay: OK, I'm ready to send the sequence right now.
>
> (Startled silence...)
>
> Bob: Uh, OK... go ahead.

Judging by Bob's reaction, he has probably never used Python or, at the very least, he has neglected to read ZeroC *Connections*.

The situation is also reversible. A third party can easily send Python code to a client and have them run it to test a server being debugged locally.

Pathological Tests

Sometimes we want to see what will happen to code if we throw situations at it that we would not normally put in a test suite. We might decide to create some transient code to see how robust a server is under improbable or impossible fringe conditions, or even out of simple curiosity. Interactive sessions or short scripts are perfect for these kinds of test.

Live System Tests, Monitoring and Ad-hoc Invocations

Ad-hoc invocations are useful for occasional informal validation and administration of a running system. A simple example might be to use the Ice object method `ice_ping()` to check if an object is alive and accessible. If a pattern emerges from repeated ad-hoc queries, they can easily be captured in a script for later use. A series of pings, or more specific queries triggered by a crontab entry can make for a cheap and efficient watchdog system.

Interpreting Servers

Running a server in the interpreter is not particularly special. However, providing interactive access to a running server can be valuable under certain conditions. In most situations, running

a server in a debugger is prohibitively intrusive or even impossible. However, running the server interactively in an interpreter is almost the same as running it normally, and it provides pretty much unlimited access to the live environment. Reasons for running a server in an interactive session might be:

- testing and debugging servers during system testing

  It is hard to have too much information when testing and debugging a program. Running the server in the interpreter in interactive mode is simply debugging the server!

- performing hot updates on running server code

  Implementation of key parts of the server can be reloaded using the methods described earlier in this article. Care must be taken to consider dependencies when updating part of a server, but it can be done. Updating the sort routine as we did above is an example.

- local monitoring and administration

  While you might be familiar with running a server in interactive mode for debugging, have you ever considered using the interpreter as a simple text mode administration tool? While not really appropriate for critical servers or servers handling sensitive information, the interpreter makes a good starting point for a very powerful text-based administrative console.

Ctrl-D

The Ice/Python combination is a potent toolset for creating distributed systems. With it, we can quickly create, test, and modify code allowing rapid evolution of a nascent system. We can even change the structure and behavior of a program while the system is running! The agility of the development system allows us to explore a problem domain quickly and experiment with code to test our assumptions. Because it is so easy to change things, there are fewer excuses for making do with low-quality code, we end up with better code and we can evolve our application smoothly. We also saw how the Python system makes it easy to debug our systems by allowing us full access to the objects in a running system.

Ice for Python can contribute greatly to the success of your next project. If you are embarking on a new project or are caught in an endless cycle of rewrites and back-tracking in your current project, you should try Ice for Python. Get out and explore!

The Grim Reaper: Making Objects Meet Their Maker

Michi Henning, Chief Scientist

Introduction

One of the most commonly used patterns in distributed systems is the *factory pattern*. Object factories are the distributed equivalent of constructors. While constructors create and initialize objects in the local address space, factories create and initialize objects in a remote address space. Whenever you use the factory pattern, you are also likely to encounter the problem of how to reclaim resources in the server if clients are misbehaved. This article outlines an approach to dealing with the problem that is both simple and effective.

Object Factories and Object Lifespan

Clients invoke operations on object factories in order to create objects in a server. Object factories are singleton objects. A typical object factory interface looks as follows:

```
// Slice
interface SomeObject
{
    // ...
};

interface SomeObjectFactory
{
    SomeObject* create(/* params... */);
};
```

The `create` operation allocates whatever resources are needed by the new object, initializes object state from the supplied parameters, and returns a proxy to the newly-created object. The flip side of the coin, namely destruction of an object, is typically provided by the object itself:

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

Once a client is finished with an object it previously created, it invokes the `destroy` operation, which reclaims any resources that are used by the object.

For many applications, the objects that a factory creates on behalf of a client have a lifespan that is limited by the lifespan of the client: the client uses the created objects in order to do something but, once the client is finished with its job or exits, the objects are no longer needed and must be destroyed. This is a common pattern for all kinds of stateful interactions between client and server. An example is a shopping cart for an on-line retail application: the cart stores the items that the client wants to order; the client destroys the cart once the customer has submitted payment details and finalized the order.

Objects In, Garbage Out

Inherent in the factory pattern is a potential problem: the server creates objects on behalf of a client (and, as a result, dedicates resources such as memory, disk space, or database connections), and the client is responsible for destroying these objects again once they are no longer needed. Unfortunately, clients have a habit of neglecting to do just that (and a lot more often than we care to admit). For example, with our on-line retail application, the customer may simply lose interest or be otherwise occupied for hours or even days. Similarly, the client may encounter a problem, such as a network failure, power loss, or simply suffer a crash. For any of these scenarios, the net result is that the client does not destroy its objects as it should, and the server is left sitting on the objects and their associated resources. Such abandoned objects are known as *garbage*.

The server is presented with something of a dilemma by garbage objects. The difficulty is not in how to remove the garbage objects (after all, the server knows how to destroy each object), but how to identify whether a particular object is garbage or not. The server *knows* when a client uses an object (because the server receives an invocation for the object), but the server does *not* know when an object is no longer of interest to a client (because a dead client is indistinguishable from a slow one).

Mechanisms that identify and reclaim garbage objects are known as *garbage collectors*. Garbage collectors are well-understood for non-distributed systems. (For example, many programming languages, such as Java and C#, have built-in garbage collectors.) Non-distributed garbage collectors keep track of all objects that are created, and perform some form of connectivity analysis to determine which objects are still reachable; any objects that are unreachable (that is, objects to which the application code no longer holds any reference) are garbage and are eventually reclaimed.

Unfortunately, for distributed systems, traditional approaches to garbage collection do not work because the cost of performing the connectivity analysis becomes prohibitively large. For example, DCOM provided a distributed garbage collector that turned out to be its Achilles' heel: the collector did not scale to large numbers of objects, particularly across WANs, and several attempts at making it scale failed.

An alternative to distributed garbage collection is to use timeouts to avoid the cost of doing a full connectivity analysis: if an object has not been used for a certain amount of time, the server assumes that it is garbage and reclaims the object. The drawback of this idea is that it is possible for objects to be collected while they are still in use. For example, a customer may have placed a number of items in a shopping cart and gone out to lunch, only to find on

return that the shopping cart has disappeared in the mean time.

Yet another alternative is to use the *evictor pattern*: the server puts a cap on the total number of objects it is willing to create on behalf of clients and, once the cap is reached, destroys the least-recently used object in order to make room for a new one. This puts an upper limit on the resources used by the server and eventually gets rid of all unwanted objects. But the drawback is the same as with timeouts: just because an object has not been used for a while does not necessarily mean that it truly is garbage.

Neither timeouts nor evictors are true garbage collectors because they can collect objects that are not really garbage, but they do have the advantage that they reap objects even if the client is alive, but forgets to call `destroy` on some of these objects.

Occam's Scythe: Reaping Objects by the Swath

Traditional garbage collection fails in the distributed case for a number of reasons:

- Garbage collectors require connectivity analysis, which is prohibitively expensive. Furthermore, for distributed object systems that permit proxies to be externalized as strings (such as Ice and CORBA), connectivity analysis is impossible because proxies can exist and travel by means that are invisible to the runtime. For example, proxies can exist as records in a database and can travel as strings inside e-mail messages.
- Garbage collectors consider all objects in existence but, for the vast majority of applications, only a small subset of all objects actually ever needs collecting. The work spent in examining objects that can never become garbage is wasted.
- Garbage collectors examine connectivity at the granularity of a single object. However, for many distributed applications, objects are used in groups and, if one object in a group is garbage, all objects in the group are garbage. It would be useful to take advantage of this knowledge, but a garbage collector cannot do this because that knowledge is specific to each application.

In the remainder of this article, we examine a simple mechanism that allows you to get rid of garbage objects cheaply and effectively. The approach has the following characteristics:

- Only those objects that potentially can become garbage are considered for collection.
- Granularity of collection is under control of the application: you can have objects collected as groups of arbitrary size, down to a single object.
- Objects are guaranteed not to be collected prematurely.
- Objects are guaranteed to be collected if the client crashes or suffers loss of connectivity.
- The mechanism is simple to implement and has low run-time overhead.

It is equally important to be aware of the limitations of the approach:

- The approach collects objects if a client crashes, but offers no protection against clients that are still running, but have neglected to destroy objects that they no longer need. In other words, the server is protected against client-side hardware failure and catastrophic client crashes, but it is not protected against faulty programming logic of clients.
- The approach is not transparent at the interface level: it requires changes (albeit minor ones) to the interface definitions for an application.
- The approach requires the client to periodically call the server, thus consuming network resources even if the client is otherwise idle.

Despite the limitations, this approach to garbage collection is applicable to a wide variety of applications and meets the most pragmatic need: how to clean up in case something goes badly wrong (rather than how to clean up in case the client misbehaves).

An Extra Level of Indirection

In the case of our object factory, the factory is a singleton object that creates objects on behalf of arbitrary clients. It is important for our garbage collector to know which client created what objects, so the collector can reap the objects created by a specific client if that client crashes. We can easily deal with this requirement by adding the proverbial level of indirection: instead of making the factory a singleton object, we provide a singleton object that creates factories. Clients first create a factory and then create all other objects they need using that factory:

```
// Slice
interface SomeObject { /* ...*/ };

interface Session
{
    SomeObject* create();
    nonmutating string getName();
    void destroy();
    idempotent void refresh();
};

interface SessionFactory // Singleton
{
    Session* create(string name);
};
```

Clients obtain a proxy to the `SessionFactory` singleton and call `create` to create a `Session` object. In turn, the `Session` object provides a `create` operation to create new objects (of type `SomeObject` in this case). Note that the `name` allows a client to distinguish different sessions (assuming the client uses more than one session). The `name` parameter is not used to identify clients or to provide a unique identifier for sessions. The `getName` operation on the `Session` object returns the name that was used by the client to create it.

Each `Session` object remembers which objects it created. Because each client uses its own `Session` object, the server knows which objects were created by what client. In normal operation, a client first creates a session, and then uses the session to create the remaining objects it needs. Once the client has finished its job, it calls `destroy` on the session. The implementation of `destroy` destroys both the session and all objects that were created by that session to reclaim resources.

To deal with crashed clients, the server needs to know when a session is no longer in use. This is the purpose of the `refresh` operation: clients are expected to periodically call `refresh` on their session objects. For example, the server might decide that, if a client's session has not been refreshed for more than ten seconds, the session is no longer in use and reclaim it. As long as the client calls `refresh` at least once every ten seconds, the session (and the objects it created) remain alive; if more than ten seconds elapse, the server simply calls `destroy` on the session. Of course, there is no need to hard-wire the timeout value—you can make it part of the application's configuration. However, to keep the implementation simple, it is useful to have the same timeout value for all sessions, or to at least restrict the timeouts for different sessions to a small number of fixed choices—this considerably simplifies the implementation in both client and server.

Server-Side Implementation

The implementation on the server side almost suggests itself:

- Whenever `refresh` is called on a session, the session records the time at which the call was made.
- The server runs a reaper thread that wakes up once every ten seconds. The reaper thread examines the timestamp of all sessions and, if it finds a session last time-stamped more than ten seconds ago, it calls `destroy` on that session.
- Each session remembers the objects it has created and destroys these objects as part of its `destroy` implementation.

Here then is the reaper thread in outline. (Note that we have simplified the code in this article to show the essentials. For example, we have omitted the code that is needed to make the reaper thread terminate cleanly when the server shuts down. You can find the full code in the `demo/Ice/session` directory in the Ice distribution.)

```
// C++
class ReapThread : public IceUtil::Thread,
    public IceUtil::Monitor<IceUtil::Mutex>
{
public:
    ReapThread();
    virtual void run();
    void add(const Demo::SessionPrx&,
        const SessionIPtr&);

private:
    const IceUtil::Time _timeout;
    struct SessionProxyPair
    {
        SessionProxyPair(
            const Demo::SessionPrx& p,
            const SessionIPtr& s) :
            proxy(p), session(s) { }
        const Demo::SessionPrx proxy;
        const SessionIPtr session;
    };
    std::list<SessionProxyPair> _sessions;
};

typedef IceUtil::Handle<ReapThread> ReapThreadPtr;
```

Note that the reaper thread maintains a list of pairs. Each pair stores the proxy of a session and its servant pointer. We need both the proxy and the pointer because we need to invoke methods on both the Slice interface and the implementation interface of the session. Whenever a client creates a new session, the server calls the `add` method on the reaper thread, passing it the new session:

```
// C++
void ReapThread::add(const SessionPrx& proxy,
    const SessionIPtr& session)
{
    Lock sync(*this);
    _sessions.push_back(
        SessionProxyPair(proxy, session));
}
```

The `run` method of the reaper thread is a loop that sleeps for ten seconds and calls `destroy` on any session that has not been refreshed for more than ten seconds:

```
// C++
void ReapThread::run()
{
    Lock sync(*this);

    while(true)
    {
        timedWait(_timeout);
        list<SessionProxyPair>::iterator p =
            _sessions.begin();
        while(p != _sessions.end())
        {
            try
            {
                //
                // Session destruction may take
                // time in a real-world example.
                // Therefore the current time
                // is computed for each iteration.
                //
                if((IceUtil::Time::now() -
                    p->session->timestamp()) >
                     _timeout)
                {
                    p->proxy->destroy();
                    p = _sessions.erase(p);
                }
```

```
                else
                {
                    ++p;
                }
            }
            catch(const
                Ice::ObjectNotExistException&)
            {
                p = _sessions.erase(p);
            }
        }
    }
}
```

Note that the reaper thread catches `ObjectNotExistException` from the call to `destroy`, and removes the session from its list in that case. This is necessary because it is possible for a client to call `destroy` explicitly, so a session may be destroyed already by the time the reaper thread examines it.

The `SessionFactory` implementation is trivial:

```
// C++
class SessionFactoryI :
    public Demo::SessionFactory
{
public:
    SessionFactoryI(const ReapThreadPtr&);
    virtual Demo::SessionPrx create(
        const std::string&, const Ice::Current&);

private:
    ReapThreadPtr _reaper;
};
```

The constructor is passed the instantiated reaper thread and remembers that thread in the `_reaper` member.

The `create` method adds each new session to the reaper thread's list of sessions:

```
// C++
SessionPrx SessionFactoryI::create(
    const string& name,
    const Ice::Current& c)
{
    SessionIPtr session = new SessionI(name);
    SessionPrx proxy = SessionPrx::uncheckedCast(
        c.adapter->addWithUUID(session));
    _reaper->add(proxy, session);
    return proxy;
}
```

Note that each session internally has a unique ID that is unrelated to its name—the name exists purely as a convenience for the application.

The server's `main` function starts the reaper thread and instantiates the session factory:

```
// C++
ReapThreadPtr reaper = new ReapThread();
reaper->start();
adapter->add(new SessionFactoryI(reaper), Ice::
stringToIdentity("SessionFactory"));
adapter->activate();
```

This completes the implementation on the server side. Note that there is very little code here, and that much of this code is essentially the same for each application. For example, we could easily turn the `ReapThread` class into a template class to permit the same code to be used for sessions of different types.

Client-Side Implementation

On the client side, the application code does what it would do with an ordinary factory, except for the extra level of indirection: the client first creates a session, and then uses the session as its factory.

As long as the client-side calls `refresh` at least once every ten seconds, the session remains alive and, with it, all objects the client created via that session. Once the client misses a `refresh` call, the reaper thread in the server cleans up the session and its objects.

To keep the session alive, you could sprinkle your application code with calls to `refresh` in the hope that at least one of these calls is made at least every ten seconds. However, that is not only error-prone, but also fails if the client blocks for some time. A much better approach is to run a thread in the client that automatically calls `refresh`. That way, the calls are guaranteed to happen even if the client's main thread blocks for some time, and the application code does not get polluted with `refresh` calls. Again, we show a simplified version of the refresh thread here that does not deal with issues such as clean shutdown and a few other irrelevant details:

```
// C++
class SessionRefreshThread :
        public IceUtil::Thread,
        public IceUtil::Monitor<IceUtil::Mutex>
{
public:
    SessionRefreshThread(
            const IceUtil::Time& timeout,
            const SessionPrx& session) :
        _session(session),
        _timeout(timeout)
    {
    }

    virtual void run()
    {
        Lock sync(*this);
        while(true)
        {
            timedWait(_timeout);
```

```
            try
            {
                _session->refresh();
            }
            catch(const Ice::Exception& ex)
            {
                return;
            }
        }
    }

private:
    const SessionPrx _session;
    const IceUtil::Time _timeout;
};

typedef IceUtil::Handle<SessionRefreshThread>
    SessionRefreshThreadPtr;
```

The only other code change in the client is to instantiate the reaper thread after creating a session. We assume that the client has a proxy to the session factory in the `factory` variable:

```
// C++
SessionPrx session = factory->create(name);
SessionRefreshThreadPtr refresh = new
SessionRefreshThread(
    IceUtil::Time::seconds(5), session);
refresh->start();
```

Note that, to be on the safe side and also allow for some network delays, the client calls `refresh` every five seconds; this is to ensure that at least one call to `refresh` arrives at the server within each ten-second interval.

Thereafter, the only other code changes in the client are to clean up the reaper thread before the client exits, which is trivial.

RIP

Reaping objects based on a session concept is a very effective way to protect a server against resource exhaustion in the face of failures, either of the clients themselves, or of the client-side hardware or network. Note that, if a client forgets to call `destroy` on an object but keeps the session alive, the object will not be reaped. However, the approach we outlined solves the hard part of the problem, namely how to do deal with catastrophic failures.

You can easily extend the approach to adapt it to your application's needs. For example, if you require different timeouts for different types of objects, you can use multiple sessions, each with a different timeout. And, by using multiple sessions, you can reduce the granularity at which objects are reaped, down to a single object. However, keep in mind that threads are a limited resource as well, so if you have many separate sessions, you should probably use a single reaper and refresh thread for several sessions. Also keep in mind that a larger number of sessions incurs a correspondingly larger amount of network traffic. However, unless you have hundreds of sessions and a very short timeout, this is not likely to be a concern.

Overall, reaping objects is remarkably non-intrusive to existing code: a few lines in the client and the server are sufficient. And, as we suggested earlier, judicious use of templates can reduce the amount of code even further. Most importantly, the additional code has no impact on the object implementations, that is, it is transparent to the bulk of the application code. This means that you can even back-patch a reaper into existing code with little effort.

We recommend that you give the Grim Reaper serious consideration in your projects: unfortunately, remote objects come with their own philosopher's stone and, unless you want them to literally live forever, you need to provide them with a tombstone to match: **RIP**†

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: *If I run any client or server that uses the Ice for C++ SSL plug-in, I get a pop-up window that says "The ordinal XXXX could not be located in the dynamic link library LIBEAY32.dll". What does this mean, and how can I fix this problem?*

You get this error when your program attempts to load an incompatible copy of the DLL. The Ice for C++ SSL plug-in, `IceSSL<version>.dll`, is linked with the OpenSSL DLLs (`libeay32.dll` and `ssleay32.dll`) and needs the correct version at run time. Often, you will get this error because another application has installed an older version of the DLLs in your Windows system directory (usually `C:\Windows\System32`). One common application with this unfortunate behavior is Intel® PROSet: the drivers for wireless Centrino network cards.

Windows looks for DLLs in the system directory before directories in your `PATH` (see http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/load_time_dynamic_linking.asp for details). Hence, if there is an old version of the OpenSSL DLLs in the Windows system directory and the correct OpenSSL DLLs are not in the same directory as your .exe, you will likely encounter this problem.

The preferred solution is to put your program and all the DLLs it needs (the Ice DLL, IceSSL DLL, OpenSSL DLLs, and so on) in the same directory. This will ensure that the proper version is used, no matter which DLL is installed elsewhere. If you deploy an application, this is by far the safest packaging. Unfortunately, putting everything in the same directory is not always practical.

Another solution is to rebuild the OpenSSL libraries with different names (e.g. libeay32-097e.dll and ssleay32-097e.dll) and rebuild Ice for C++ using these names. This is very reliable, but requires more build effort.

Yet another solution is to have a closer look at these conflicting DLLs: which program actually uses them? Do you really need this program? (For example, recent releases of Intel PROSet continue to install these DLLs but do not appear to load them.) If the DLLs are indeed used, we recommend Sysinternals Process Explorer to identify the process or processes loading these DLLs. If they are indeed essential, you may try to move them to another directory (such as `C:\Windows\System32\openssl`) and add this directory to the system `PATH` before the directory containing the Ice OpenSSL DLLs with the same names. In this way, the program that depends on these old OpenSSL DLLs likely will continue to work; for Ice applications, add the directory with the new OpenSSL DLLs to your `PATH` before running them.

Q: How do I configure my Ice server for NAT?

First we should create a sample network configuration that we can use for this discussion. We'll assume that your firewall has the IP address 123.4.1.1 and that your server host is in a private network with the address 192.168.0.5. The firewall is configured to forward all network traffic from its port 7000 to port 9999 on your server's host.

If your server's object adapter is named `LoginAdapter`, you can configure its endpoint with the property shown below:

```
LoginAdapter.Endpoints=tcp -h 192.168.0.5 -p 9999
```

This configuration is sufficient to allow the server to receive port-forwarded requests from the firewall, but there is a potential problem: proxies created by this object adapter contain the server host's private IP address and port, which are inaccessible to clients on the other side of the firewall. In order for a client to communicate with your server, the client must use a proxy that contains the firewall's address and port.

This is not an insurmountable problem. In simple situations, where a client only uses one proxy to communicate with the server, the client can bootstrap a proxy containing the firewall's address in a number of ways, such as by calling `stringToProxy` or reading it from a file. However, when the server creates proxies dynamically, the object adapter requires additional configuration.

An object adapter actually has two sets of endpoints: the physical endpoints on which it listens for requests, and the published endpoints that appear in the proxies it creates. If no published endpoints are defined, then the physical endpoints are used by default. The example above illustrates why it isn't always appropriate to publish the object adapter's physical endpoints.

To correct the problem, we define an additional property:

```
LoginAdapter.PublishedEndpoints=tcp -h 123.4.1.1
-p 7000
```

Now all of the proxies created by the object adapter will advertise the firewall's address and port.