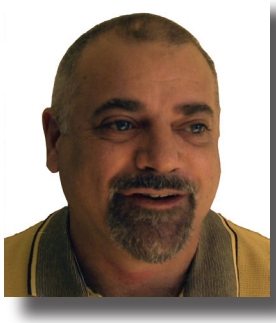
Connections

ZeroC's Newsletter for the Ice Community

Issue Number 11, February 2006



The Nice Thing about Standards…

Recently, another raging debate broke out in comp.object.corba about the value of standards. The standard in question is DDS (which Bernard wrote about in Issue 6 of *Connections*). DDS is actually the fifth (yes, *fifth*) specification published by the OMG for event distribution (not counting a sixth attempt for a real-time notification service that never went anywhere). These specifications are: the original event service, the lightweight event service, the notification service, the unreliable multi-cast specification, and DDS.

Why so many of these standards, all of which are meant to do the same thing, namely, to provide publish-subscribe decoupled communications? The answer can be found in history. The original event service, published in 1994 (and revised in minor ways in 2001) turned out to be severely dysfunctional. In particular, its pull model makes it pretty much impossible to build robust and scalable implementations of the service, and its lack of topic-based event distribution requires applications to be aware of the channel topology, which creates an administrative and maintenance nightmare. And, to top it off, a large part of the specification, namely the typed event service, was so under-specified that no-one ever implemented it.

The lightweight event service (published in 2004) is essentially the same as the event service, but with the problematic pull model removed. However, it retains the awkward channel-based distribution as well as the typed event APIs.

The notification service, first published in 2000 and revised in 2002, improves on the event service by adding filtering capabilities and structured events (in a somewhat clumsy attempt at providing an alternative to the typed event service). However, despite the known problems, the notification service retained both the pull model and the typed event APIs. The 2002 revision fixed quite a few defects, including a major one, namely, that the original specification contained illegal IDL: the specification could not be compiled with a conforming IDL compiler and had obviously never been fully implemented prior to publication. (One wonders about the obvious procedural problems of an international standards organization that publishes unimplementable standards…)

Another issue with the notification service was that implementations performed rather poorly: the requirements on filtering and quality-of-service imposed by the specification make it essentially impossible to create a scalable and high-performance implementation.

In response to these performance problems, the OMG published the unreliable multi-cast specification for event distribution. True, multi-cast does indeed perform very well. However, it is pragmatically useless: on LANs under moderate load, many events are lost; on WANs, even more events are lost (disregarding the fact that most routers will not forward multi-cast packets anyway, meaning that *all* events are lost).

The latest incarnation of the OMG's event distribution service, DDS, is supposed to address these problems. Whether it will remains to be seen. What we do know is that the specification requires *several hundred* lines of IDL definitions, and that implementations from different vendors do not interoperate until the DDS interoperability specification is finally published and implemented.

In the mean time, OMG customers get to choose among five specifications that, respectively, do not scale and have poor performance, lose a prohibitively large number of events, or do not interoperate. I'm reminded of Andy Tanenbaum's famous quote about standards: "The nice thing about standards is that there are so many to choose from."

Michi Henning
Chief Scientist

Issue Features

Dynamic Ice, Part 1: Efficient Request Forwarding

In the first of a two part series on dynamic programming with Ice, Mark Spruiell introduces the untyped invocation and dispatch interfaces for efficiently forwarding Ice invocations.

Freeze Indexes

Matthew Newhook continues his article series by showing you how to use Freeze indexes for efficient database access.

Contents

Dynamic Ice, Part 1

Mark Spruiell, Senior Software Engineer

Dynamic Ice, Part 1: Efficient Request Forwarding

Some Ice features don't normally attract much attention, as is the case with the "Dynamic Ice" APIs. What we call Dynamic Ice is really a collection of related APIs that allows you to manipulate data in the Ice encoding format as well as handle Ice requests without knowledge of the operation's parameters. Most applications don't need these capabilities, but when you do need it, you *really* need it. The APIs can be separated into two categories:

- streaming
- untyped invocation and dispatch

The streaming API encodes the values of Slice types into a sequence of bytes, and vice versa. The untyped invocation and dispatch API makes it possible to send and receive Ice requests in a generic fashion. These two APIs can be used independently of one another. For example, the streaming API might come in handy when you need to store a persistent representation of a Slice type in a database, and the untyped invocation and dispatch API is extremely useful for event distribution and routing applications. Of course, you can also combine the APIs, which is exactly how the Ice extensions for Python and PHP work.

We'll explore the streaming API in a future article. For now, let's focus on the functionality provided by untyped invocation and dispatch.

The Man Behind the Curtain

If you've ever peeked at the generated code produced by your Slice definitions, you'd know that a lot of work is being done behind the scenes whenever you invoke an operation on a proxy. Your application code appears to be making a simple function call, and of course our goal is to make distributed object computing with Ice as familiar as possible. In response to a proxy invocation, however, the generated code for the operation encodes the parameters in their proper order and hands this data off to the Ice run time, which prepares and sends a request message in the format defined by the Ice protocol.

A request message consists of header information, such as the object identity and the operation name, followed by a block of data containing the parameters as encoded by the generated code. Similarly, a reply message contains the encoded form of the output parameters and return value, which the generated code knows how to interpret.

You can think of the generated code as a typed wrapper around the untyped Ice core: the core treats the request and reply data as arbitrary sequences of bytes. The core doesn't care what's contained in these "blobs" of bytes, its only responsibility is to deliver them safely and efficiently. The information in the message header enables the core to process these messages without needing to examine their operation-specific payloads.

A Bit of History

Early on in Ice's development, the need for a secure connection concentrator became readily apparent. The program would serve as the front-end for a collection of servers, accepting connections from hundreds or thousands of clients and forwarding their requests over a single connection to each back-end server. Thus, the Glacier router was born, the predecessor to our current Glacier2 service.

Just like a network router device, the Glacier router's duty was to forward messages to their intended destinations as quickly as possible. To that end, the router needed the ability to forward a message without decoding and re-encoding it. Not only would this have a severe impact on its throughput, but it would also require application-specific knowledge. That is, the router would need to know the parameter signature of each operation, and therefore would need to be updated each time a Slice definition changed. This requirement would impose other limitations as well, such as the inability to support multiple versions of an application simultaneously.

Fortunately, the Ice encoding makes it possible to forward messages without wasting precious time decoding and re-encoding them. (See Michi's editorial for information on another distributed computing technology that cannot make this claim.) The API that allowed Glacier, and later IceStorm, to perform this task was present in early Ice releases but remained undocumented for some time while we evaluated its usability.

Introducing the Blobject

`Blobject` is the name of the Ice superclass for a special kind of servant that intercepts the normal request dispatching process. In a regular servant, which derives from the generated class corresponding to a Slice interface, the untyped message is interpreted by the inner workings of the generated code and transformed into a method call on the implementation class. A `Blobject` servant skips this step and exposes the untyped message via the `ice_invoke` method:

```
// C++
namespace Ice
{
class Blobject : virtual public Object
{
public:
    virtual bool
    ice_invoke(
        const vector<Byte>& inParams,
        vector<Byte>& results,
        const Current& curr) = 0;
};
};
```

As you can see, the input parameters are provided as a blob, and the operation's results are expected in the same form. If the servant needed to interpret the input parameters or manually construct the results, it would use the aforementioned streaming API, but of course the servant must know exactly how each blob is structured. A general purpose request-forwarding application typically considers these blobs to be opaque.

It's important to understand that all of the requests received by this servant are passed to `ice_invoke`. The last argument, having the type `Ice::Current`, provides a wealth of information about the request, including the identity of the target object and the name of the operation. A `Blobject` servant generally uses the members of `Ice::Current` to determine how it should process the request.

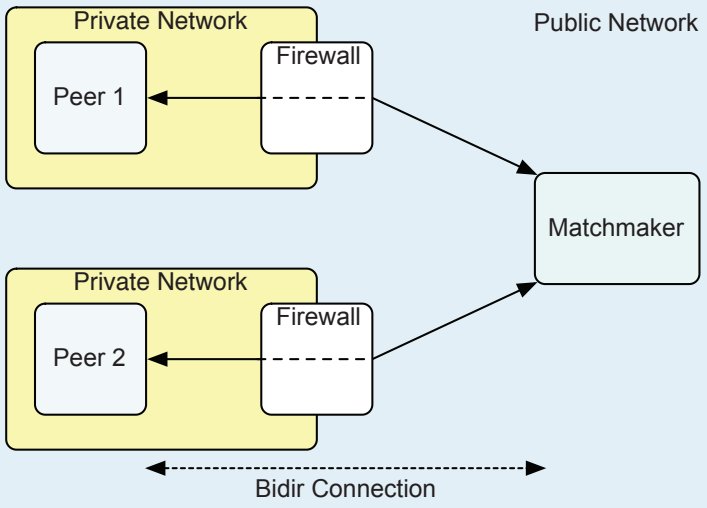
The `ice_invoke` method is expected to return true if the request succeeded. In the case of a user exception, the method must return false and supply the encoded exception in results. The method may also throw subclasses of `RequestFailedException`, such as `ObjectNotExistException`.

Since `Blobject` derives from `Object`, you can use a `Blobject` servant anywhere a regular servant is accepted. For example, you can populate an object adapter's active servant map with a combination of `Blobject` and regular servants; the adapter does not distinguish between them. More often though, a `Blobject` servant is used as a default servant in conjunction with a servant locator, enabling a single servant to process the requests of any number of target objects.

Matchmaker

We'll use an object registry service to demonstrate the implementation of a `Blobject` servant. The service enables peers that reside behind firewalls to advertise and locate objects. Some peers will act as servers and register their objects with the service, while others will obtain proxies for those objects and invoke requests on them. The registry is acting as a forwarding service, blissfully unaware of the interfaces supported by the advertised objects as well as the contents of the messages it is passing along. To circumvent firewall restrictions, the peers that advertise objects must establish bidirectional connections to the registry so that requests targeted at their objects can be forwarded back to them. Figure 1 shows the relationship between the registry and the peers.

Figure 1: Matchmaker



The Slice interface for our registry is quite simple:

```
// Slice
#include <Ice/Identity.ice>

module Demo
{
interface Registry
{
    void add(Ice::Identity id);
    void remove(Ice::Identity id);
    Object* locate(Ice::Identity id);
};
};
```

A peer invokes add to inform the registry that an object with the given identity is available and can be reached via the peer's bidirectional connection with the registry. When the object is no longer available, the peer must call remove to notify the registry. Finally, peers use `locate` to find registered objects. The `locate` operation returns a nil proxy if no match was found; otherwise, the peer can downcast the returned proxy to the appropriate interface and invoke operations on it as usual. The proxy returned by `locate` actually refers to our `Blobject` servant, which transparently forwards the request to its intended recipient and returns the results.

The complete source code for the registry service is available as a separate download.

Initial Design

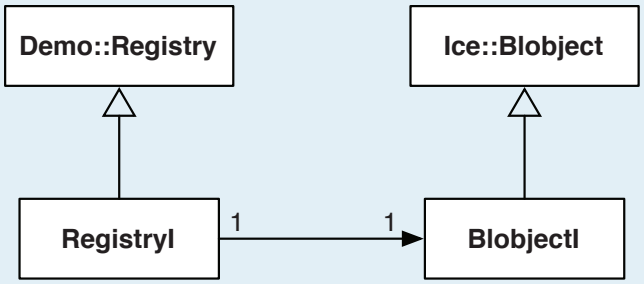
We know we want to use a `Blobject` servant in the registry, but there are several approaches we could take. For example, we could add a new servant to the object adapter's active servant map for each object in the registry. This one-to-one relationship between objects and servants is easy to understand and implement, and would work fine as long as the number of active objects doesn't grow very large. However, we can create a more scalable design

without much additional effort by using a default servant and a servant locator.

The plan is to add our registry servant to the active servant map and install a servant locator. Invocations directed at the registry's identity will be handled by the registry servant; invocations on all other identities result in a query to the servant locator, which always returns our `Blobject` servant to handle the requests.

The `Blobject` servant will need to maintain a map that associates identities with proxies. In each call to `ice_invoke`, the servant must access the map to obtain the proxy it can use to forward the message. The operations on the `Registry` interface represent manipulations of this map, so our implementation of the registry interface will simply delegate to the `Blobject` servant, as illustrated by the class diagram in Figure 2.

Figure 2: Class Diagram



Registry Servant

To reinforce the relationship shown in Figure 2, let's begin our examination of the sample application with the registry servant. It's little more than a wrapper around the `Blobject` servant:

```
// C++
class RegistryI : public Demo::Registry
{
public:
    RegistryI(const BlobjectIPtr& blobject)
        : _blobject(blobject) {}

    virtual void
    add(const Identity& id, const Current& curr)
    {
        _blobject->add(id, curr.con);
    }

    virtual void
    remove(const Identity& id, const Current&)
    {
        _blobject->remove(id);
    }

    virtual ObjectPrx
    locate(const Identity& id, const Current&)
    {
        return _blobject->locate(id);
    }

private:
    BlobjectIPtr _blobject;
};
```

Notice that the implementation of `add` passes a member of `Current` to the `Blobject` in addition to the identity. The `con` member represents the incoming connection from the peer to the registry service.

The peer must configure its connection to our service for bidirectional use, which allows the registry to forward requests over the incoming connection. In the absence of a bidirectional connection, the registry would need to be able to establish a new connection to the peer in order to deliver a request. If that was possible, the need for a separate forwarding service is greatly reduced, since it means peers could communicate directly with one another. For the sake of discussion, we'll assume that firewalls prevent peers from establishing direct connections to each other, and therefore bidirectional connections are a necessity.

Blobject Servant

Now that we understand the relationship between the registry servant and the `Blobject` servant, we can move on to the `Blobject` implementation:

```
// C++
class BlobjectI : public Blobject, public Mutex
{
public:
    BlobjectI(const ObjectAdapterPtr&);

    virtual bool
    ice_invoke(
        const vector<Ice::Byte>& inParams,
        vector<Ice::Byte>& results,
        const Current& curr);

    void
    add(const Identity& id,
        const ConnectionPtr& conn);
    void
    remove(const Identity& id);
    ObjectPrx
    locate(const Identity& id);

private:
    ObjectAdapterPtr _adapter;
    map<Identity, ObjectPrx> _objects;
};
typedef Handle<BlobjectI> BlobjectIPtr;
```

The class derives from both `Blobject` and `IceUtil::Mutex`; the latter is necessary so that we can safely access the map from multiple threads. For the convenience of our registry servant, we've used the `IceUtil::Handle` template to define a smart pointer for our `Blobject` implementation class.

Administrative Tasks

The `add`, `remove` and `locate` methods in our servant correspond directly to the administrative operations of the `Registry` interface. Let's examine those first, then we'll proceed to `ice_invoke`.

In the implementation of `add`, we acquire a lock and then check to see if the map already contains an entry for the given identity. If so, we remove it, and then add a new association between the identity and a proxy. As previously discussed, the registry servant supplies the `Connection` object so that `add` can create a proxy that uses the peer's bidirectional connection.

```
// C++
void
BlobjectI::add(
    const Identity& id, const ConnectionPtr& conn)
{
    Lock lock(*this);
    map<Identity, ObjectPrx>::iterator p =
        _objects.find(id);
    if(p != _objects.end())
    {
        _objects.erase(p);
    }
    _objects[id] = conn->createProxy(id);
}
```

Figure 3 shows the map that `add` updates as each object is announced. In the diagram, Peer 1 hosts the objects O1 and O2, while Peer 2 hosts the objects O3 and O4. The proxies P1 through P4 were created by the bidirectional connections from Peer 1 and Peer 2 and are stored in the map using the object identities as keys.

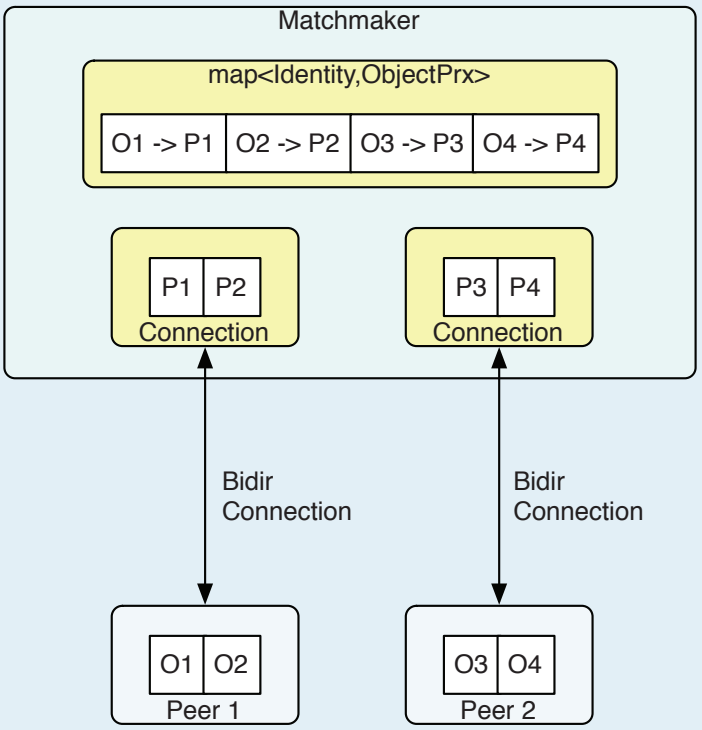
The implication with all bidirectional connections is that they can only be used as long as the connection remains active. For this reason, we must disable active connection management in our peers to prevent idle connections from being automatically closed by the Ice run time.

The `remove` method simply disassociates the identity from a proxy:

```
// C++
void
BlobjectI::remove(const Identity& id)
{
    Lock lock(*this);
    map<Identity, ObjectPrx>::iterator p =
        _objects.find(id);
    if(p != _objects.end())
    {
        _objects.erase(p);
    }
}
```

The `locate` method looks so simple that its importance might be overlooked. The caller of the equivalent registry operation expects to receive a proxy that will enable it to invoke operations on the desired target object. It would be tempting to return the proxy that we find in the map, but that is not the correct proxy. This proxy was created in the `add` method to allow the servant to communicate with the target object, but a remote peer could not use this proxy. In fact, the Ice run time would raise an exception if we tried to return that proxy because it can only be used in the address space of the registry service and thus cannot be marshaled. (Refer to the Ice manual for more information on the semantics of proxies created by bidirectional connections.)

Figure 3: Adding an item to the registry



The correct proxy needs to contain the endpoints of the `Blobject` servant's object adapter so that all invocations on the proxy are sent to our servant. Therefore, we invoke `createProxy` on the object adapter and pass the target identity:

```
ObjectPrx
BlobjectI::locate(const Identity& id)
{
    Lock lock(*this);
    map<Identity, ObjectPrx>::iterator p =
        _objects.find(id);
    if(p != _objects.end())
    {
        return _adapter->createProxy(id);
    }
    return 0;
}
```

Implementing ice_invoke

Now that we've covered the administrative duties of our service, let's discuss `ice_invoke`. To minimize the amount of time we

hold the lock, we obtain the proxy for the target object in a nested block. Assuming a match was found, we first ensure that we are using the appropriate facet name, and then we forward the request. Finally, if no match was found for the target identity, `ice_invoke` raises `ObjectNotExistException`.

```
// C++
bool
BlobjectI::ice_invoke(
    const vector<Byte>& inParams,
    vector<Byte>& results, const Current& curr)
{
    ObjectPrx proxy;

    {
        Lock lock(*this);
        map<Identity, ObjectPrx>::iterator p =
            _objects.find(curr.id);
        if(p != _objects.end())
        {
            proxy = p->second;
        }
    }

    if(proxy)
    {
        if(!curr.facet.empty())
        {
            proxy = proxy->ice_newFacet(
                curr.facet);
        }
        // forward request here ...
        return ...;
    }
    ObjectNotExistException ex(
        __FILE__, __LINE__);
    ex.id = curr.id;
    ex.facet = curr.facet;
    ex.operation = curr.operation;
    throw ex;
}
```

Forwarding a Request

Before we can write the line of code that's incomplete in `ice_invoke` shown above, we need to discuss the API for forwarding a request. The base proxy class (`ObjectPrx`) defines a method, not coincidentally named `ice_invoke`, that allows us to do just that:

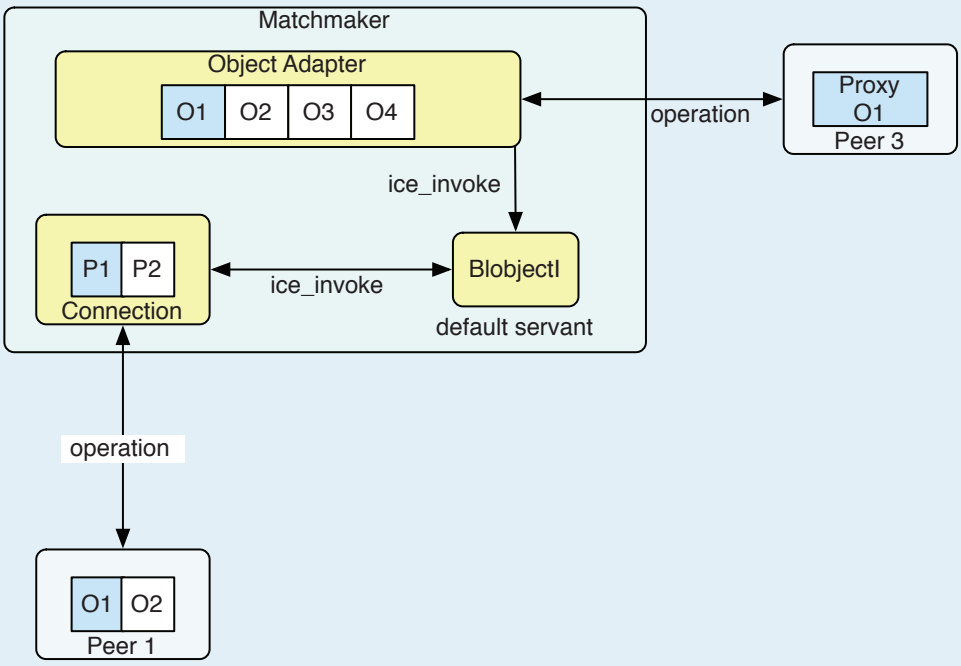
```
// C++
bool ice_invoke(
    const string& name,
    OperationMode mode,
    const vector<Byte>& inParams,
    vector<Byte>& results);
```

The return value of this method has the same semantics as in `Blobject`: a value of true indicates success, whereas false indicates the occurrence of a user exception.

We have all of the data we need in our servant to complete this request, so the complete line of code becomes the following:

```
// C++
return proxy->ice_invoke(
    curr.operation, curr.mode, inParams, results,
                curr.ctx);
```

Figure 4: Request from Peer 3 to Peer 1



The operation's name, mode and context are obtained from the servant's `Current` argument. The return value of the proxy's `ice_invoke` method becomes the return value of the servant's `ice_invoke` method. User exceptions are handled automatically, and local exceptions such as `OperationNotExistException` pass unhindered back to the caller.

Figure 4 shows a request as it travels from one peer to another via the registry service. Peer 3 has used locate to obtain a proxy for the object O1. When Peer 3 invokes an operation using the proxy, the request is delivered to our `Blobject` servant as a call to `ice_invoke`. The servant retrieves the proxy P1 from its map; this proxy corresponds to the identity O1 of the target object. Next, the servant calls `ice_invoke` on proxy P1, forwarding the request over the bidirectional connection to Peer 1 and returning the results to Peer 3.

Core Operations

All Ice objects support a few core operations:

- `string ice_id()`
- `StringSeq ice_ids()`
- `bool ice_isA(string id)`
- `void ice_ping()`

These operations are described in the Ice manual, but they need to be mentioned because a `Blobject` servant might encounter them. As mentioned earlier, a `Blobject` receives *all* operations invoked on the object, including these core operations. A regular servant derived from a generated class normally ignores these operations because their implementations are provided by a base class, but that is not the case with `Blobject`. You must decide whether these operations are relevant for your application and respond appropriately. In the case of the registry service, these operations are forwarded to the target object just like all other operations. Under different circumstances, an entirely different response might be necessary.

For example, the IceStorm publish-subscribe service supports N-to-N relationships between publishers and subscribers. It would not make sense for IceStorm to forward operations such as `ice_id`, `ice_ids` and `ice_isA` to subscribers, for a number of reasons. First, there may not be any subscribers at the time the operation is invoked. Or there might be several subscribers, each of which might return a different response to the same request, so selecting one at random would not provide a meaningful answer. Finally, IceStorm only forwards oneway requests, so all of the core operations except `ice_ping` are ineligible anyway.

In other cases, the forwarding service might be able to respond on behalf of the target object without needing to consult it directly.

Lastly, you might decide not to support these operations at all, in which case you should consider raising an exception such as `OperationNotExistException` in order to notify the caller. Be aware that eliminating support for `ice_isA` means that clients of your service cannot use checked casts to narrow their proxies.

Registry Server

The implementation of the servant locator is so trivial that I will refer you to the accompanying source archive for the (meager) details. However, the main program for the registry service is worth a brief look. We'll use the `Service` helper class, which makes it easy to run our server as a Windows service or Unix daemon (see my article in issue 10 issue of *Connections* for more information on this class). Here's the class definition:

```
// C++
class RegistryService : public Service
{
public:
       RegistryService();
protected:
       virtual bool start(int, char*[]);
};
```

The `start` method is where our server does all of its initialization work:

```
// C++
bool
RegistryService::start(int, char*[])
{
    ObjectAdapterPtr adapter = communicator()->
        createObjectAdapter("Registry");

    BlobjectIPtr blobj = new BlobjectI(adapter);
    RegistryPtr registry = new RegistryI(blobj);
    adapter->add(registry,
        stringToIdentity("registry"));
    ServantLocatorPtr locator =
        new LocatorI(blobj);

    adapter->addServantLocator(locator, "");
    adapter->activate();

    return true;
}
```

After creating the object adapter and the `Blobject` servant, we add our registry servant to the active servant map. Next we create and register the servant locator; using an empty string as its category marks it as the default servant locator.

Notice that our registry servant and the servant locator are installed on the same object adapter. Given the rules by which an adapter searches for a servant in response to a request, an invocation on the identity registry will always be directed to our registry servant. The implication is that a peer must never attempt to register that identity.

There are a couple of things we could do about this situation. One possibility is to choose an identity for the registry servant that is less likely to conflict with a peer's. Or we could create two object adapters, one to host the registry servant and one for the servant locator and `Blobject` servant. The disadvantage of creating another object adapter is that it adds another endpoint that peers must be able to reach. For our purposes, we can accept the need for a single "reserved" identity.

For the sake of completeness, here's our `main` function, which simply instantiates the service and blocks until terminated:

```
// C++
int
main(int argc, char* argv[])
{
    RegistryService service;
    return service.main(argc, argv);
}
```

Sample Peer

Our registry service is able to forward invocations on Ice objects of any type, so let's create a simple peer to exercise the `Blobject` servant. We'll use a variation of our classic hello demo as the Slice interface for our application:

```
// Slice
module Demo
{
interface Hello
{
    void sayHello(string from);
};
};
```

Our implementation of the `Hello` interface does what you'd expect:

```
// C++
class HelloI : public Hello
{
public:

    virtual void
    sayHello(const string& from,
        const Current&)
    {
        cout << from << " says hello." << endl;
    }
};
```

We use the `Application` helper class to simplify our peer:

```
// C++
class Client : public Application
{
public:
    virtual int run(int, char*[]);
};
```

Finally, the notable parts of the `run` method are shown below. You can find the complete source code in the corresponding source archive for this article.

The peer accepts two command-line arguments. The first argument is the identity of the Ice object hosted by the peer. The optional second argument is the identity of another object that we'll locate and, if found, invoke `sayHello` on it.

```
  // C++
int
Client::run(int argc, char* argv[])
{
       string name = argv[1];
       string peer;
       if(argc > 2)
       {
              peer = argv[2];
       }
}
```

Next, we create an object adapter and activate our servant:

```
// C++
ObjectAdapterPtr adapter = communicator()->
    createObjectAdapterWithEndpoints(
        "Client", "tcp");
Identity ident = stringToIdentity(name);
adapter->add(new HelloI, ident);
adapter->activate();
```

We're ready to use the registry, so we set up the bidirectional connection and invoke `add` on the registry to announce our object:

```
// C++
RegistryPrx registry = ...;
registry->ice_connection()->setAdapter(adapter);
registry->add(ident);
```

It's important to establish the bidirectional connection before calling `add`, as we have shown above. Doing it in the opposite order would leave open an (admittedly small) window of opportunity for another peer to locate our object and attempt to invoke an operation on it before the bidirectional connection was properly established, resulting in an exception.

If the identity of a peer was supplied on the command line, we try to locate the object using the registry and then pass our own name to `sayHello`:

```
// C++
if(!peer.empty())
{
    ObjectPrx peerProxy =
    registry->locate(stringToIdentity(peer));
    if(peerProxy)
    {
        HelloPrx hello =
            HelloPrx::checkedCast(peerProxy);
        hello->sayHello(name);
    }
    else
    {
        cerr << appName() << ": no peer found"
        <<  " matching `" << peer << "'" << endl;
        return EXIT_FAILURE;
    }
}
```

Instructions for running the sample application are provided in a README file in the source archive.

Threading Issues

In its current form, the registry service suffers from a few problems. With a default configuration, the service uses the thread pool concurrency model with a single thread in each of the client and server thread pools. As a result, the service would deadlock while forwarding the first request. To understand the reason for the hang, you need to be familiar with the responsibilities of the two thread pools.

The client thread pool is responsible for all messages received from an outgoing connection. This normally consists of processing replies to outstanding requests, but it also includes dispatching incoming requests if the connection is configured for bidirectional use.

Similarly, the server thread pool handles incoming connections, dispatching requests to servants but also processing replies to outgoing requests sent on a bidirectional connection.

Now let's review the sequence of events that causes the service to hang:

1. a request arrives for a target object
2. the request is received by the (only) thread in the server thread pool
3. that same thread forwards the request on the bidirectional connection to the target object
4. the target object completes the operation and returns the reply
5. the service hangs because the reply is not processed; the server thread pool is responsible for processing this reply, but its only thread is blocked waiting for the outgoing request to complete.

We could avoid this situation by increasing the maximum size of the server thread pool in the service, and that might be sufficient in a carefully controlled environment. However, when the maximum number of simultaneous peers is unknown, you must either set the maximum size of the thread pool to be so large that, if it ever actually reached that size, the service might run out of resources, or you must set the maximum size to a reasonably large value and hope that it never grows that large. To prevent a deadlock, we must ensure that there is always at least one thread available in the server thread pool to process replies.

A second problem is the way in which the service is forwarding requests. As we saw in the sequence of events leading up to the deadlock, a thread from the server thread pool is blocked while the forwarded request completes. Not only is this a waste of a scarce resource, but it also makes the service vulnerable to misbehaving or malicious peers. For example, suppose that the target object becomes blocked in its implementation, caused by a programming error or a conflict while attempting to acquire some resource. The service's thread is now blocked indefinitely. If several peers attempt to invoke requests on this object, the number of available threads in the server thread pool will shrink rapidly and could eventually result in the same deadlock we experienced before.

It was exactly these types of problems that prompted the addition of the thread-per-connection concurrency model. The Glacier router needed to be robust in the face of such challenges, and the thread-per-connection model helps in this regard. Using this model, a new thread is dedicated to each incoming and outgoing connection. Although it's less scalable than a thread pool, the thread-per-connection model is preferable for this particular application because it isolates each peer from the activities of other peers, thereby eliminating the possibility of resource starvation.

We can change our registry service to use the thread-per-connection model instead of thread pool with a simple change to its configuration properties. We have now avoided a potential deadlock, as long as a peer does not attempt to invoke an operation on one of its own objects. If a peer did try to call one of its own objects via the service, the service's thread associated with the peer's connection would deadlock in a sequence of events similar to that described above. To remedy this limitation, we'll need some new tools.

Asynchronous Blobject

Our threading issues primarily revolve around the fact that our `Blobject` servant forwards its requests using synchronous nested invocations, blocking the thread until the nested invocation completes. We can improve this situation by using the asynchronous API for untyped invocation and dispatch.

A different superclass, `BlobjectAsync`, enables a `Blobject` servant to receive requests with asynchronous semantics. The class definition is shown below:

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

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

Like any asynchronous method dispatch (AMD) operation, the `ice_invoke_async` method receives an AMD callback object, the input parameters (as a blob), and the `Current` argument. For the Ice run time to consider the request complete, the implementation is required to invoke one of the methods on the callback object, at which point a reply message is sent back to the caller. Note that the Ice run time does not require the request to complete before `ice_invoke_async` returns. For example, the implementation of `ice_invoke_async` could store the request information in a queue that is processed by a separate worker thread. As the thread finishes each task, it invokes the stored callback object to complete the request and send the reply.

As you might expect, the base proxy class `ObjectPrx` provides an equivalent method for sending untyped invocations asynchronously:

```
// C++
void ice_invoke_async(
    const AMI_Object_ice_invokePtr& callback,
    const string& operation,
    OperationMode mode,
    const 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 comprising 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<Ice::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 data.

In the case of an exception such as `ObjectNotExistException`, the Ice run time calls `ice_exception` and passes the exception instance.

Request Chaining

The definition of our `Blobject` servant requires only minor changes to use the asynchronous API. We change its superclass from `Blobject` to `BlobjectAsync`, its name from `BlobjectI` to `BlobjectAsyncI`, and replace `ice_invoke` with `ice_invoke_async`:

```
// C++
class BlobjectAsyncI :
    public BlobjectAsync, public Mutex
{
public:
    // ...
    virtual void
    ice_invoke_async(
        const AMD_Object_ice_invokePtr& cb,
        const vector<Ice::Byte>& inParams,
        const Current& curr);
    // ...
```

The changes to the servant's implementation are more interesting. The servant receives an AMD callback object, but we cannot invoke it until the forwarded request completes. Furthermore, we have to supply a callback object to the proxy method `ice_invoke_async` when forwarding the request. Our solution is to store the AMD callback object in the AMI callback object. Here's the definition of our AMI 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:

    AMD_Object_ice_invokePtr _cb;
};
```

As you can see, the callback is fairly simple. The `ice_response` and `ice_exception` methods delegate to the AMD callback object. We call this technique "request chaining." If you stop to think about it, this solution makes perfect sense: the AMI callback object knows when the forwarded request completes, so it is the most logical entity to complete the AMD request.

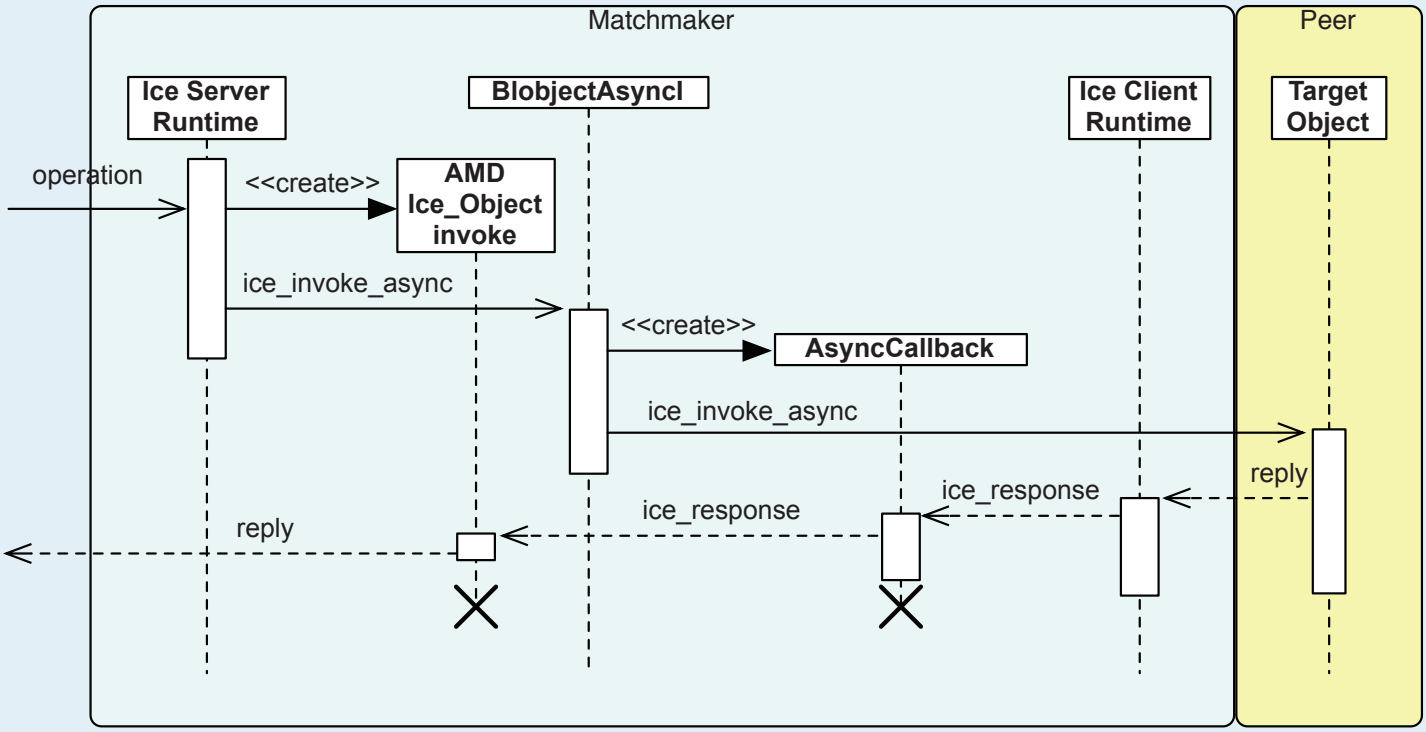
Now we're ready to discuss the new `ice_invoke_async` method. You'll notice that it looks largely the same as its synchronous version, but there are subtle differences. After obtaining the correct proxy, the method creates an AMI callback object, passing the AMD callback object to its constructor to save for later use. Next, the method forwards the request by calling `ice_invoke_async` on the proxy, and lets the AMI callback object notify the AMD callback object when the request completes. Figure 5 shows these interactions as a request is forwarded using the asynchronous APIs.

The `return` statement following the call to `ice_invoke_async` is important so that we don't fall through to the code that raises `ObjectNotExistException`. In this case, the exception isn't actually raised, but rather returned to the caller via the AMD callback object. (We did this for demonstration purposes; the Ice run time also allows you to raise these exceptions directly.)

```
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())
```

Figure 5: Request Chaining




```
    {
            proxy = p->second;
        }
    }

    if(proxy)
    {
        if(!curr.facet.empty())
        {
            proxy = proxy->ice_newFacet(
                curr.facet);
        }
        AMI_Object_ice_invokePtr amiCallback =
            new AsyncCallback(amdCallback);
        proxy->ice_invoke_async(
            amiCallback, curr.operation,
            curr.mode, inParams, curr.ctx);
        return;
    }

    ObjectNotExistException ex(
        __FILE__, __LINE__);
    ex.id = curr.id;
    ex.facet = curr.facet;
    ex.operation = curr.operation;
    amdCallback->ice_exception(ex);
}
```

That's all there is to it. The combination of thread-per-connection and asynchronous request chaining makes our registry service much more robust. Admittedly, we have increased the complexity of the implementation somewhat, but the effort was worthwhile.

For another example of using asynchronous APIs and request chaining, see issue 4 of Connections.

Future Directions

There are several ways you could extend the registry service to add more functionality.

One possibility is a store-and-forward feature: if a request arrives for an object that is registered but not currently active, the service could save the header information and parameter blob until the object becomes available. Realistically, this feature would need to be implemented using the asynchronous API. Callers must also be prepared to wait for a response if the object is inactive.

Security is another area for potential improvement. At present, any peer can request a proxy for any registered object. More restrictions might be necessary in a real-world situation.

Finally, it might be useful to have other ways to search for an object of interest. An efficient solution is to change the `add` operation to accept metadata that describes the object and add an operation that queries this metadata for one (or more) matching objects.

Freeze Indexes

Matthew Newhook, Senior Software Engineer

The previous article in this series showed how to use Freeze maps with the introduction of a user registry service. This article demonstrates the use of Freeze indexes and ranged searches in both Java and C++.

What is an Index?

An index allows the application to do an efficient lookup by a value other than a Freeze dictionary's primary key. The initial implementation of the user manager in Issue 7 of Connections used a Freeze index to do a reverse lookup of the username to the user proxy. I recommend that you review this article before continuing. The primary need for an index is efficiency: it is possible to implement the queries supported by an index without it, but it is very inefficient because it requires a linear search of all the database records, whereas an index performs lookups in $O(\log n)$ time.

Imagine that you want to implement a query to find all inactive users—that is all users who have not logged into the chat server for some period of time. To implement this, we must record a time stamp when a user logs into the chat server. Where does this time stamp belong? We could place it into the user record that is maintained by the user manager. However, if we do this, to implement the query, we must contact all user managers and then consolidate the query results returned by each manager. To avoid this, we can put the time stamp into the user registry. Let's first review the user registry interface:

```
// Slice
interface UserRegistry
{
    void add(string name, User* proxy)
        throws UserExistException;
    void remove(string name)
        throws UserNotExistException;
    nonmutating User* find(string name);

    void checkConsistency();
};
```

The user registry maintains a Freeze map called `StringUserPrxDict` that maps a user name to a user proxy. Since we will now also maintain a time stamp, we'll define a struct as follows:

```
// Slice
struct UserInfo
{
    User* proxy;
    string username;
    long lastLoginTime;
};
```

The `lastLoginTime` member is the time in seconds since January 1, 1970 UTC. We've also added a field for the user name because callers frequently need this information; adding the field avoids the need for an additional RPC. The Freeze map now maps a user name to a `UserInfo`.

FreezeScript Transformation

First, we'll write a database transformation using FreezeScript so that we can transform the old database to the new Slice format.

We have changed a Freeze map of `string–User*` pairs to a Freeze map of `string–UserInfo` pairs. The `proxy` field of the `UserInfo` will contain the `value` field of the old map. What about the `username` and `lastLoginTime` field? Clearly we cannot compute a meaningful value for `lastLoginTime`, since this information was not previously stored. Therefore, we'll initialize this field to 0 (which is the default value that FreezeScript assigns to a long).

How about the `username` field? Can't we use the map key for this? As the implementation currently stands, we can because the unaltered user name is used as the database key. However, this is actually a bug in the previous implementation: it allows two user names to differ only in case, which should not be permitted. As such, we'll fix this bug in the code and proceed in this section as if the bug did not exist.

The user name can only be looked up via a call on the user object, which is not possible during a transform. Therefore, we need to write code to fill in the `username` field. When is a good time to run this code? With the current implementation, the best time is during a database consistency check: if the `username` field is empty, we'll fill in the value. After an administrator runs the database transform and restarts the chat server, a database consistency check should be run immediately. The code change to the consistency check is as follows:

```
// C++
StringUserInfoDict::iterator p = dict.begin();
while(p != dict.end())
{
    try
    {
        if(p->second.username.empty())
        {
            UserProfile profile =
                p->second.proxy->getProfile();
            UserInfo info = p->second;
            info.username = profile.userId;
            p.set(info);
        }
        else
        {
            p->second.proxy->ice_ping();
        }
    }
    catch(const ObjectNotExistException&)
    {
```

```
        dict.erase(p++);
        continue;
    }
    catch(const Exception&)
    {
        // Ignore
    }
    ++p;
}
```

Ok, now the transform itself. This is done as follows:

```
// FreezeScript XML
<transformdb>
  <database key="string"
   value="::Chat::User*,::Chat::UserInfo">
    <record>
      <set target="newvalue.proxy"
       value="oldvalue"/>
    </record>
  </database>
</transformdb>
```

The `database` element says that they map key is a `string` and does not change. The old map value is a `::Chat::User*` and the new map value is a `::Chat::UserInfo`. The `record` element defines several variables, including `newvalue` (the new database value) and `oldvalue` (the old database value). As described above, we want to assign `oldvalue` to the `proxy` field.

Ok, now let's test this. First start the chat server demo from Issue 10. Start the chat server, add two users foo and foo2, and then terminate the server. Now dump the contents of the old database. These instructions assume that the current working directory is the demo for this article, and that the old demo resides in ../chat.old.

```
may$ dumpdb -I. --load ../chat.old/UserRegistry.ic
e --key string --value ::Chat::User* ../chat.old/d
b/node/servers/UserRegistry/dbs/UserRegistry userr
egistry
Key: 'foo'
Value: ::Chat::User*("user/c0:a8:1:68:3526cf:108fb
0e7532:-7fff" -t @ UserManager.UserManager.UserMan
ager)
Key: 'foo2'
Value: ::Chat::User*("user/c0:a8:1:68:3526cf:108fb
0e7532:-7ffe" -t @ UserManager.UserManager.UserMan
ager)
```

As you can see, there are two records in the database: `foo` and `foo2`. Next, start the `icegridnode` and use `icegridadmin` to deploy the chat server—this creates the correct directory structure in the `db` directory. Once this is done, shut down the server and transform the database. The FreezeScript XML is contained in a file called `migrate.xml`.

```
may$ transformdb --include-old ../chat.old --old
../chat.old/UserRegistry.ice --include-new . --new
UserRegistry.ice -f migrate.xml ../chat.old/db/nod
e/servers/UserRegistry/dbs/UserRegistry userregist
ry db/node/servers/UserRegistry/dbs/UserRegistry
warning: unable to transform from ::Chat::User* to
::Chat::UserInfo
warning: unable to transform from ::Chat::User* to
::Chat::UserInfo
```

Now dump the contents of the new database

```
may$ dumpdb -I. --load UserRegistry.ice --key stri
ng --value ::Chat::UserInfo db/node/servers/UserRe
gistry/dbs/UserRegistry userregistry
Key: 'foo'
Value: struct ::Chat::UserInfo
{
    proxy = ::Chat::User*("user/c0:a8:1:68:3526cf:
108fb0e7532:-7fff" -t @ UserManager.UserManager.Us
erManager)
    username = ''
    lastLoginTime = long(0)
}
Key: 'foo2'
Value: struct ::Chat::UserInfo
{
    proxy = ::Chat::User*("user/c0:a8:1:68:3526cf:
108fb0e7532:-7ffe" -t @ UserManager.UserManager.Us
erManager)
    username = ''
    lastLoginTime = long(0)
}
```

Great! That's exactly what should be there. Next, copy the old user manager database environment (this doesn't need a transform) into the new database directory for the user manager.

```
may$ cp ../chat.old/db/node/servers/UserManager/db
s/UserManager/* db/node/servers/UserManager/dbs/Us
erManager
```

Start the chat server, and run a consistency check. This can be done with the admin tool.

```
may$ ./admin
commands
add <user> <password
password <user> <password
delete <user>
findInactive
findByRange t1 t2
check
quit
==> check
checking user registry consistency... ok!
checking user manager consistency.... ok!
==> quit
```

Now dump the database contents again:

```
may$ dumpdb -I. --load UserRegistry.ice --key stri
ng --value ::Chat::UserInfo db/node/servers/UserRe
gistry/dbs/UserRegistry userregistry
Key: 'foo'
Value: struct ::Chat::UserInfo
{
```

```
    proxy = ::Chat::User*("user/c0:a8:1:68:3526cf:
108fb0e7532:-7fff" -t @ UserManager.UserManager.Us
erManager)
    username = 'foo'
    lastLoginTime = long(0)
}
Key: 'foo2'
Value: struct ::Chat::UserInfo
{
    proxy = ::Chat::User*("user/c0:a8:1:68:3526cf:
108fb0e7532:-7ffe" -t @ UserManager.UserManager.Us
erManager)
    username = 'foo2'
    lastLoginTime = long(0)
}
```

As you can see, the `username` field has been correctly filled in.

C++ Freeze Indexes

For the C++ version, we need to change the `slice2freeze` command as follows:

```
slice2freeze -I. -I<slice-dir> --dict Chat::String
UserInfoDict,string,Chat::UserInfo StringUserInfoD
ict UserRegistry.ice
```

Now we'll add a new method to notify the user registry that a user has logged in.

```
// Slice
sequence<UserInfo> UserInfoSeq;
interface UserRegistry
{
    // …
    User* login(string user);
};
```

The implementation without error checking and deadlock recovery is as follows:

```
// C++
UserPrx
UserRegistryI::login(const string& username, const
Current&)
{
    ConnectionPtr connection =
        createConnection(_communicator, _name);
    StringUserInfoDict dict(
        connection, "userregistry");
    StringUserInfoDict::iterator p =
        dict.find(username);
    if(p != dict.end())
    {
        UserInfo info = p->second;
        info.lastLoginTime =
            Time::now().toSeconds();
        p.set(info);
        return p->second.proxy;
    }
    return 0;
}
```

Next, we need to add a method to find all inactive users. We can do this by adding a method that does a ranged search returning all values that fall between two login times.

```
// Slice
sequence<UserInfo> UserInfoSeq;
interface UserRegistry
{
    // …
    nonmutating UserInfoSeq findByRange(
        long t1, long t2);
};
```

Now how can we implement this? Without an index, we would have to use something like the following (once again, without error checking or deadlock recovery):

```
// C++
UserInfoSeq
UserRegistryI::findByRange(
    Long lowerBound, Long upperBound,
    const Current&) const
{
    ConnectionPtr connection =
        createConnection(_communicator, _name);
    StringUserInfoDict dict(
        connection, "userregistry");
    UserInfoSeq l;
    StringUserInfoDict::const_iterator p;
    for(p = dict.begin(); p != dict.end(); ++p)
    {
        if(p->second.lastLoginTime >= lowerBound&&
            p->second.lastLoginTime < upperBound)
        {
            l.push_back(p->second);
        }
    }
    return l;
}
```

This implementation is very inefficient because it iterates over all records in the database. With a Freeze index, we can make this far more efficient. To generate the index, we must pass the `--dict-index` option to `slice2freeze` as follows:

```
--dict-index Chat::StringUserInfoDict,lastLoginTi
me
```

The `--dict-index` option extends the implementation of the Freeze map by adding the following member functions to the generated class:

```
// C++
iterator findByLastLoginTime(Long, bool = true);
iterator lowerBoundForLastLoginTime(Long);
iterator upperBoundForLastLoginTime(Long);
std::pair<iterator, iterator>
equalRangeForLastLoginTime(Long);
int lastLoginTimeCount(Long) const;
```

In addition, `slice2freeze` adds `const` versions of the first four functions. These methods match the `find`, `upper_bound`,

lower_bound, equal_range and count methods defined by the C++ standard for an associative collection. In a nutshell, they work as follows:

- findByLastLoginTime returns an iterator to the first element in the map that matches the parameter value. If the boolean parameter is false, you get an iterator to the first element in the map that matches the parameter value; if the parameter is true, the returned iterator will only return those elements that match the parameter. (This boolean is only present for backwards compatibility purposes, and I recommend that you use equalRangeForLastLoginTime instead.)
- lowerBoundForLastLoginTime returns an iterator to the first element in the map that has a value greater than or equal to the given parameter value.
- upperBoundForLastLoginTime returns an iterator to the first element in the map with a key greater than the parameter value.
- equalRangeForLastLoginTime returns a pair of iterators. The first points to the first element in the map that contains the given parameter value, and the second to the element just after the last element that contains the parameter value.
- lastLoginTimeCount returns the number of elements that match the given parameter.

For our lastLoginTime use case, the findByLastLoginTime, equalRangeForLastLoginTime and lastLoginTimeCount are not very useful methods because searching for a precise timestamp is uncommon. The default collation order used by an index is based on the Ice binary encoding (and sorting by binary little-endian encoding of an Ice::Long is not very meaningful). We need the data sorted by the actual lastLoginTime value, which we can do as follows:

```
--dict-index Chat::StringUserInfoDict,lastLoginTi
me,sort
```

The third argument specifies a sort function. By default, the comparator used for sorting is std::less<MEMBER_TYPE>. If we want a map to be sorted in most-recently-logged-in order instead, we can use:

```
--dict-index Chat::StringUserInfoDict,lastLoginTim
e,sort,std::greater<long>
```

For our use case we want the map in least-recently-logged-in order, so this is just what we need.

With that out of the way, findInactive can be implemented as follows (once again, no error checking or deadlock recovery):

```
// C++
UserInfoSeq
UserRegistryI::findByRange(
    Long lowerBound, Long upperBound,
    const Current&) const
{
    ConnectionPtr connection =
        createConnection(_communicator, _name);
    StringUserInfoDict dict(
        connection, "userregistry");
    UserInfoSeq l;
    StringUserInfoDict::const_iterator q =
     dict.upperBoundForlastLoginTime(upperBound);
    StringUserInfoDict::const_iterator p;
    for(p = dict.lowerBoundForlastLoginTime(
            lowerBound); p != q; ++p)
    {
        l.push_back(p->second);
    }
    return l;
}
```

This is much more efficient because the sorting and ordering is done by the database, meaning that only the records of interest are extracted.

To find all inactive users in C++, the method could be called as follows:

```
// C++
UserRegistryPrx registry = …;
// 30 days ago
const Long upperBound =
    (Time::now() - Time::seconds(
        60 * 60 * 24 * 30)).toSeconds();
UserInfoSeq user = registry->findByRange(
    0, upperBound);
```

To make the findByRange method somewhat more tolerant of user error, we'll ensure that the lower bound is less than the upper bound and swap if necessary. (Without the swap, findByRange would fail if lowerBound >= upperBound.)

```
// C++
UserInfoSeq
UserRegistryI::findByRange(
    Long lowerBound, Long upperBound,
    const Current&) const
{
    // …
    if(lowerBound != upperBound)
    {
        //
        // Swap the values if necessary.
        //
        if(upperBound < lowerBound)
        {
            Long tmp = upperBound;
            upperBound = lowerBound;
            lowerBound = tmp;
        }
    }
    // …
}
```

Java Freeze Indexes

As with C++, the Java Freeze implementation is also based on the Java standard API. For Freeze indexes, we make use of the `java.util.SortedMap` interface.

The first step is to set up the `slice2freezej` command in the `build.xml` ant file. As before, we have a dictionary named `StringUserInfoDict` that uses a string key and `UserInfo` value. The dictionary has an associated index using the `lastLoginTime` member of the dictionary's value type.

```
// Ant XML
<slice2freezej casesensitive="on"
outputdir="${generated.dir}">
  <includepath>
    <pathelement path="."/>
    <pathelement path="${slice.dir}"/>
  </includepath>
  <fileset dir="." includes="UserRegistry.ice"/>
  <dict name="Chat.StringUserInfoDict"
   key="string" value="::Chat::UserInfo"/>
  <dictindex name="Chat.StringUserInfoDict"
   member="lastLoginTime"/>
</slice2freezej>
```

The `dictindex` element adds the following methods to the generated code:

```
public Freeze.Map.Iterator
    findByLastLoginTime(long);
public int lastLoginTimeCount(long);
```

- `findByLastLoginTime` returns an iterator to the first element in the map that matches the given parameter value.
- `lastLoginTimeCount` returns a count of elements that match the given parameter value.

Unlike the C++ version, the sort order for the index must be provided by the implementation, not through an option to the `slice2freezej` compiler. This is accomplished through a map that associates the index name to a `java.util.Comparator`. For our implementation, this is done as follows:

```
// Java
java.util.Map indexComparators =
    new java.util.HashMap();
final java.util.Comparator less =
new java.util.Comparator()
{
    public int compare(Object o1, Object o2)
    {
        if(o1 == o2)
        {
            return 0;
        }
        else if(o1 == null)
        {
            return -((Comparable)o2).
                compareTo(o1);
        }
        else
        {
            return ((Comparable)o1).compareTo(o2);
        }
    }
};
indexComparators.put("lastLoginTime", less);
```

When we create the Freeze map, we pass the comparator map as a parameter:

```
// Java
Freeze.Connection connection =
    Freeze.Util.createConnection(
        _communicator, _name);
StringUserInfoDict dict =
    new StringUserInfoDict(connection,
        "userregistry", true, null,
        indexComparators);
```

It is possible to add or remove a comparator, but you must not change the semantics of a comparator. For example, you cannot change the above less-than comparator to have greater-than semantics—if you do, you will suffer the consequences!

The Java Freeze map implements the `java.util.SortedMap` interface, the details of which can be viewed at http://java.sun.com/j2se/1.4.2/docs/api/java/util/SortedMap.html. In addition, the Freeze map defines a number of non-standard methods.

```
// Java
public SortedMap mapForIndex(String indexName);
```

This returns a `SortedMap` that uses the sort order as defined by the corresponding comparator in the comparators map passed to the constructor of the Freeze map.

```
// Java
public java.util.SortedMap
headMapForIndex(String indexName, Object toKey);
public java.util.SortedMap
tailMapForIndex(String indexName, Object fromKey);
public java.util.SortedMap
subMapForIndex(String indexName, Object fromKey,
    Object toKey);
```

Each of these methods has the same semantics as `headMap`, `tailMap`, and `subMap` as described in the `SortedMap` documentation, except that the sort order is determined by the comparator for `indexName`.

There is an important difference in the Java API compared to the C++ API: with the C++ API, the iterators returned by the various index methods iterate on the main map, with an order defined by the sort order of the index—therefore, duplicate index values present no difficulty.In contrast, the Java API returns a totally new map. This map uses the index value as the key, meaning that for a single key there may be multiple values. Therefore, the `SortedMap` maps an index value to a `java.util.Set` of `java.util.Map.Entry`, each `java.util.Map.Entry` being an entry contained in the main map.

Next, let's examine the implementation of the `findByRange` method. (Once again, we omit the error checking and deadlock recovery code.)

```
// Java
public UserInfo[]
findByRange(long lowerBound, long upperBound,
            Current current)
{
    List l = new ArrayList();
    if(lowerBound != upperBound)
    {
        //
        // Swap the values if necessary.
        //
        if(upperBound < lowerBound)
        {
            long tmp = upperBound;
            upperBound = lowerBound;
            lowerBound = tmp;
        }
        Freeze.Connection connection =
            Freeze.Util.createConnection(
                _communicator, _name);
        StringUserInfoDict dict =
            new StringUserInfoDict(
                connection, "userregistry", true,
                null, _indexComparators);
        SortedMap sm = dict.subMapForIndex(
             "lastLoginTime",
             new Long(lowerBound),
             new Long(upperBound));
        Iterator p = sm.values().iterator();
        while(p.hasNext())
        {
            Set values = (Set)p.next();
            Iterator q = values.iterator();
            while(q.hasNext())
            {
                Map.Entry entry =
                    (Map.Entry)q.next();
                l.add((UserInfo)entry.getValue());
            }
        }
        connection.close();
    }
    return (UserInfo[])l.toArray(new UserInfo[0]);
}
```

This implementation uses the `subMapForIndex` method to return a `java.util.SortedMap` that contains the elements in the range `lowerBound` to `upperBound`.

Note that when opening a connection or a database with Freeze for Java, it is very important to close the database and connection when done with it. If you do not do this, they will not be released until the garbage collector runs, thus keeping resources open far longer than necessary. Note that the code above only closes the connection since closing the connection closes all associated dictionaries.

The complete implementation in Java can be seen in the source distribution accompanying the newsletter.

There you have it! I hope the article makes the new index sorting facilities introduced in Ice 3.0 clearer. If you have any questions please ask on the forums and we'll do our best to help you.

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: Why do I not get concurrent invocations in a server?

By default, the Ice server-side run time uses a thread pool to dispatch incoming requests. The number of requests that can execute concurrently in a server is limited to the number of threads in the pool. If more clients attempt to concurrently call operations than there are threads in the pool, the corresponding requests are not dispatched until a currently executing invocation completes and returns its thread to the pool; that thread then picks up the next pending request.

By default, the server-side thread pool has a size of one, meaning that only one operation can execute in the server at a time. If you don't see concurrent invocations in a server, it is likely that the server is running with a thread pool containing only a single thread, thereby serializing all incoming invocations.

The size of the server-side thread pool is controlled by a number of properties:

- `Ice.ThreadPool.Server.Size`
- `Ice.ThreadPool.Server.SizeMax`
- `Ice.ThreadPool.Server.SizeWarn`

The `Ice.ThreadPool.Server.Size` property controls the number of threads in the pool. When you create a communicator, the specified number of threads are created and added to the pool; the size of the pool never drops below this number of threads.

The `Ice.ThreadPool.Server.SizeMax` property has a default value that equals the size of the thread pool. However, you can set this property to a value that is larger than `Ice.ThreadPool.Server.Size`. If you do, the server-side run time will allow the thread pool to grow in size up to this value if enough requests arrive concurrently. The run time also dynamically shrinks the thread pool back to its initial size as demand on threads is reduced (with some hysteresis to avoid continuously creating and destroying threads).

Finally, the `Ice.ThreadPool.Server.SizeWarn` property sets a threshold. If the number of threads in use exceeds this value, the run time emits a warning via the communicator's logger. The default value of this property is 80% of the value specified by `Ice.ThreadPool.Server.SizeMax`.

Q: Why do I not get concurrent replies to asynchronous invocations?

Just as there is a server-side thread pool, there is a corresponding client-side thread pool. The client-side thread pool is used to dispatch replies to asynchronous invocations (AMI) and to process requests received via bidirectional connections.

Like the server-side pool, the client-side pool has a default size of one. If your client sends several asynchronous invocations and you expect to receive replies to these invocations concurrently but, in reality, all the replies are serialized, it is likely that the client-side thread pool is at its default size of one.

The size of the client-side thread pool is controlled by the properties `Ice.ThreadPool.Client.Size`, `Ice.ThreadPool.Client.SizeMax`, and `Ice.ThreadPool.Client.SizeWarn`. The defaults and meaning of these properties is analogous to the ones for the server-side pool.