Connections

ZeroC's Newsletter for the Ice Community

Issue Number 23, May 2007

Stable!

In Issue 19 of Connections, I remarked that there seem to be few computing topics that cannot be learned in ten minutes, as attested by books with titles such as *Teach Yourself Linux in 10 Minutes*. A cursory look through the computer section at a bookshop reveals many more titles along the same lines, such as *CORBA for Dummies*, *The Complete Idiot's Guide to XML*, and *Visual C++ in 12 Easy Lessons*. I find such titles amusing because of their oxymoronic nature. No matter what one might think of middleware, dummies definitely won't be able to use CORBA, and no matter what one might think of SOAP, complete idiots won't ever get very far with XML. (Not to mention that someone who learns C++ in twelve easy lessons will, at best, know just enough to be dangerous.) In fact, these books remind me of magazine ads with titles such as *Lose 10kg in Two Weeks!* Clearly, the promise is vacuous—someone who did indeed lose that much weight in two weeks would likely be near death.

The computing industry seems to be unique in this respect, and other industries appear to be more restrained. For example, doctors do not read *Teach Yourself Surgery in 10 Minutes*, civil engineers cannot refer to *Bridge Design for Dummies*, law libraries do not stock *The Complete Idiot's Guide to Contract Law*, and one reason why planes crash so rarely might be that *Air Traffic Control in 12 Easy Lessons* is unavailable.

I see a lot of feature lists for middleware and other software. Among features such as *fast* and *small*, these lists often contain the term *stable*. I would have expected this to go without saying: *of course* the software is stable, otherwise it would not be for sale, right? Yet, few people seem to see anything remarkable in this. We are so used to software that has bugs, barely performs its advertised function, is impossibly difficult to use, or does not work at all that we are pleasantly surprised when we *do* come across a product that works. In turn, "it works" is something that marketing people won't hesitate to point out, completely missing the irony inherent in that statement. By comparison, I cannot recall ever seeing a battery torch that promised on its packaging *Lights up every time you flick the switch!* or a car brochure that proudly proclaimed *Brakes engage whenever you step on the pedal, always!*

These anecdotes highlight that we are in an industry that is immature and has barely begun to take itself seriously. Snake oil is the order of the day, and existing wisdom, hard-earned research results, and prior art are routinely ignored. The catch-cry of *I've never designed a protocol before, but how hard can it possibly be?* begets SOAP, and the quest for the holy grail of loose coupling bestows us with web services. Never mind that, in the long run, we know that these things are so flawed that they will be abandoned. In the meantime, an entire industry can continue to spin its wheels, sell silver bullets, and, whenever necessary, replace the current fad with the next one, much like going from the grapefruit diet to the tomato-and-egg diet.

If the industry does not take itself seriously, little wonder then that customers don't either. Most customers are suspicious of anything new and are tired of the pot of gold at the end of the rainbow, especially when their business depends on the reliability of their software. If we want to be taken seriously as an industry, we had better change that and start selling software that is stable as a matter of course. Here at ZeroC, we are doing just that, so please forgive us for not including a *Stable!* sticker when you buy your Ice license.

Michi

Michi Henning
Chief Scientist,

Issue Features

Proxies

In this article, Matthew Newhook describes one of the fundamental building blocks of Ice applications—proxies.

Master–Slave Replication with Ice

Benoit Foucher explains how to create replicated services using a master–slave architecture.

Contents

Proxies

Matthew Newhook, Senior Software Engineer

Proxies are a fundamental concept of Ice and, in order to write Ice applications, you need understand what proxies are, why they are necessary, and how to use them. This article provides an overview of Ice proxies and their semantics.

Overview

What is a Proxy?

The American Heritage Dictionary defines the term proxy as follows:

1. A person authorized to act for another; an agent or substitute.
2. The authority to act for another.
3. The written authorization to act in place of another.

The first definition, "agent or substitute" is the one that best captures the purpose of proxies in Ice: a proxy is the local (client-side) ambassador for a remote (server-side) Ice object. In order for an application to invoke an operation on a remote Ice object, it must have a proxy. Instead of invoking the operation directly on the remote object, the code invokes a corresponding operation on the proxy; the proxy then takes care of forwarding the invocation to the remote Ice object.

Proxies direct invocations to Ice objects. Although the purpose of this article is not to discuss Ice objects (this is the subject of another article), it is impossible to discuss proxies without having some understanding of Ice objects. Here is a simple definition:

An *Ice object* is an abstraction that has an *interface* and a unique *object identity*. An Ice object is composed of one or more *facets*. Each facet has exactly one most-derived interface. (Two or more facets of an Ice object can have the same interface.) If an Ice object has facets, the object identity is shared by all facets. Each facet has a *facet name*. No two facets of the same Ice object can have the same facet name. An Ice object may (but need not) have a *default facet*. The name of the default facet is the empty string.

How are Proxies Used?

Without proxies there would be no way for an application to send a message to an Ice object. By way of illustration, consider the following C++ code:

```
// C++
class Hello
{
public:
    void sayHello()
    {
        cout << "Hello world!" << endl;
    }
};
// ...
Hello* p = ...;
p->sayHello();
```

Calling `sayHello` via the pointer `p` sends the `sayHello` message to the `Hello` object. Let's now translate this example into an equivalent one that uses Ice. First we define the `Hello` interface in Slice:

```
// Slice Hello.ice
module Demo
{
interface Hello
{
     void sayHello();
};
};
```

Next, we compile this definition with the `slice2cpp` compiler, which generates a number of type definitions and their implementations. (See the Ice Manual for details on the C++ mapping.) One of the generated types is called `HelloPrx`, which is a C++ class. To use it, we can write the following code:

```
// C++
HelloPrx p = ...;
p->sayHello();
```

This example looks remarkably similar to the original code, with the exception that we have used the proxy class `HelloPrx` instead of a pointer to the `Hello` class. What exactly then is this `HelloPrx` class? An instance of this class is a smart-pointer to a reference-counted proxy to an Ice object that provides the `Hello` interface. In other words, an instance of the proxy class is the local C++ object through which a programmer can invoke an operation on a (possibly remote) Ice object.

What Information Does a Proxy Contain?

Consider again the simple C++ code I presented earlier:

```
// C++
Hello* p = ...;
p->sayHello();
```

What information does the variable `p` contain? Seeing that `p` is an ordinary C++ class instance pointer, it contains the address of a `Hello` instance. (The address serves as the object identity.) As for regular class instance pointers (or *references*, in languages such as Java and C#), an Ice proxy also contains addressing information. However, because a proxy can denote an object in a remote address space, the addressing information is more complex. At a minimum, a proxy contains:

- The object identity of the Ice object denoted by the proxy.
- Addressing information to locate the server(s) that implement the Ice object.

This only makes sense: the additional information identifies which server implements the Ice object, and the object identity determines which object within that server a proxy denotes.

Proxies are not opaque objects that contain hidden information. If you know the identity of an Ice object, its location, and the protocol(s) supported by the server, you can create a proxy out of thin air from a string. For example, if you know an Ice object has the identity `hello` and runs on host 192.168.1.4 at port 10000 using the TCP protocol, you can provide a stringified proxy to `stringToProxy`, which turns a string into a proxy:

```
// C++
HelloPrx p = HelloPrx::checkedCast(
    communicator->stringToProxy(
        "hello:tcp -h 192.168.1.4 -p 10000"));
p->sayHello();
```

I will discuss stringified proxies in more detail later in this article.

A proxy can contain additional settings that influence its behavior. Some of these settings are marshaled when a proxy is sent over the wire. These are:

- the facet name;
- a security setting that is used to force a proxy to make invocations only over secure connections;
- a proxy mode, which is one of twoway, oneway, batch oneway, datagram, or batch datagram.

Other proxy settings are local to the proxy and are not marshaled when the proxy is sent over the wire. These are:

- whether the proxy optimizes collocated invocations;
- a connection caching policy;
- an endpoint selection policy;
- a locator proxy;
- a locator cache timeout;
- a router proxy;
- a security policy that determines whether invocations on the proxy prefer secure connections over insecure connections;
- a default `Ice::Context` to use when making invocations;
- whether the proxy uses the thread–per-connection concurrency model;
- which connection ID to use;
- a timeout for invocations;
- whether to use compression.

Note that proxies are strongly typed (at least, in languages such as C++ or Java, which provide strong typing), and that the type of a proxy is a programming-language concept: when proxies are marshaled, neither the type of the proxy nor the type of the target object are sent over the wire.

Proxies are Immutable

Once created, a proxy becomes immutable, that is, its contents cannot be changed. If you need a proxy that is identical to an existing proxy except for one of the proxy settings, you must create a new proxy. For example, suppose you need a proxy with a particular timeout. You can set this timeout by calling `ice_timeout`:

```
// C++
HelloPrx proxy = ...;
proxy = HelloPrx::uncheckedCast(
    proxy->ice_timeout(5000));
```

The call to `ice_timeout` creates a new proxy that is identical to the source proxy, except for the new timeout of five seconds. The `uncheckedCast` is necessary because `ice_timeout` returns a proxy of type `ObjectPrx`, which is the ultimate base type of all proxies; the cast narrows the returned proxy to the (derived) type `HelloPrx`. Be aware of the following mistake:

```
// C++
HelloPrx proxy = ...;
proxy->ice_timeout(5000); // ERROR!
```

This does not change the timeout on the proxy; instead, it returns a new proxy with a five-second timeout, but that proxy is immediately thrown away!

Proxy Comparison

You can compare proxies for equality. By default, proxy comparison compares all aspects of a proxy, including the object identity, facet name, addressing information, and all the proxy settings; two proxies compare equal only if they are identical in all respects. This is often not what is intended:

```
// C++
HelloPrx h1 = ...:
HelloPrx h2 = HelloPrx::uncheckedCast(
    h1->ice_timeout(5000)); // Set a new timeout
assert(h1 == h2); // Assertion fails
```

This assertion will fail, as the two proxies have different timeout values. Usually, what is needed is to find out whether two proxies denote the same Ice object. To do this, you need to compare the object identities:

```
// C++
HelloPrx h1 = ...;
HelloPrx h2 = HelloPrx::uncheckedCast(
    h1->ice_timeout(5000)); // Set a new timeout
assert(h1->ice_getIdentity() ==
    h2->ice_getIdentity()); // Assertion passes
```

The reason that comparing only the object identities also compares the Ice objects is that, as previously stated, the Ice object model assumes that every Ice object has a unique identity. Therefore, if the identities in the proxies are the same, so are the Ice objects denoted by the proxies.

All of the Ice language mappings provide convenience functions for proxy comparison. For C++, the convenience function to compare object identities is `proxyIdentityEqual`. For example:

```
// C++
HelloPrx h1 = ...;
HelloPrx h2 = HelloPrx::uncheckedCast(
    h1->ice_timeout(5000)); // Set a new timeout
// Assertion passes
assert(proxyIdentityEqual(h1, h2));
```

This code is equivalent to the preceding example (which extracted the identities and then compared the identities explicitly).

Sometimes it is necessary to compare object identity and facet name, to determine whether two proxies denote the same facet of the same Ice object. In C++, the convenience function to do this is `proxyIdentityAndFacetEqual`. (Please consult the Ice Manual for the equivalent methods for other language mappings.)

Slice Proxies

Consider the following C++ code:

```
// C++
class Widget { };
class WidgetFactory
{
     Widget create();
};
```

Compare this to:

```
// C++
class WidgetFactory
{
     Widget* create();
};
```

The `create` method on the first widget factory returns a widget, whereas the `create` method on the second widget factory returns a *pointer* to a widget. Now consider the following Slice:

```
// Slice
class Widget
{
};
interface WidgetFactory
{
      Widget create();
};
```

Compare this to:

```
// Slice
interface WidgetFactory
{
     Widget* create();
};
```

Much like C++, the first `create` operation returns a widget by value, whereas the second `create` operation returns a *proxy* to a widget. Thus, the Slice syntax `Widget*` means "return a *proxy* to a widget". This is often a point of confusion for developers new to Ice. As with C++, passing a class by value is entirely different from passing a pointer to a class. Passing a class by value (the first form) passes all of the data associated with the class and creates a new instance initialized with the class data in the receiver's address space, whereas passing a proxy to a class passes only the proxy and leaves the class instance where it is.

Proxy Types

Proxies come in several varieties. All of them contain the identity of the associated Ice object and information such as a timeout, plus additional information that varies with the type of proxy.

Direct Proxies

Direct proxies contain a protocol identifier (such as TCP, UDP, or SSL) and addressing information for that protocol, that is, the host and port at which the server runs. Together with the object identity, this is sufficient to contact the target object.

Indirect Proxies

Indirect proxies contain no addressing information—to contact the Ice object, the client-side run time first obtains the addressing information using an Ice location service, such as IceGrid. (For more information on IceGrid, see Michi's article "IceGrid in 10 Minutes" in Issue 19 of *Connections*.)

Indirect proxies have two forms. The first is known as a *well-known proxy* that contains only the identity of an Ice object. The client-side run time obtains the actual addressing information for such a proxy by asking the location service for the direct proxy of an object with that identity. (Once known, the resolved proxy is cached by the Ice run time for later use.) The following example contacts a well-known proxy with the identity `hello`:

```
// C++
HelloPrx p = HelloPrx::checkedCast(
    communicator->stringToProxy("hello"));
p->sayHello();
```

The second form of indirect proxies contains the object identity and an *object adapter identifier*. The client-side run time obtains the actual addressing information for such a proxy by asking the location service for the addressing information of the corresponding object adapter. The code below contacts an Ice object with identity `hello` that is hosted by an object adapter with the adapter identifier `HelloAdapter`:

```
// C++
HelloPrx p = HelloPrx::checkedCast(
    communicator->stringToProxy(
        "hello@HelloAdapter"));
p->sayHello();
```

Note that both direct and indirect proxies may additionally be routed. Routed proxies do not contact their target Ice object directly, but instead send all invocations to their configured router. Routers can be used to build forwarding services such as Glacier2. (See Michi's article "Glacier2 in 10 Minutes" in Issue 22 of *Connections* for more information on Glacier2.)

Fixed Proxies

Fixed proxies are bound to a particular connection for the entire life time of a proxy. Once that connection is closed, the proxy no longer works (and will never work again). In addition, fixed proxies cannot be marshaled. Fixed proxies are used for bi-directional communications to allow servers to call back to an object provided by the client without opening a separate outgoing connection from server to client.

Proxy Methods

Ice proxies provide a number of methods. What follows is a listing of the available methods and examples of their use. (As always, see the Ice Manual for a complete list of these methods.)

Remote Inspection

These methods return information about the associated Ice object. For remote objects, they will therefore make a remote invocation.

```
// Pseudo Slice
bool ice_isA(string id);
void ice_ping();
StringSeq ice_ids();
string ice_id();
```

For example:

```
// C++
ObjectPrx obj = communicator()->stringToProxy(
        "Hello:tcp -h 192.168.1.4 -p 10000");
assert(obj->ice_isA(Hello::ice_staticId());
```

The `ice_isA` method determines whether the associated Ice object implements the given interface and returns true if so; false otherwise. The generated `ice_staticId` method returns the type ID of the given interface. If the Ice object is not reachable, `ice_isA` throws an exception.

Local Inspection

These methods inspect the configuration and state of the proxy. The methods never make an invocation on the target object and, therefore, do not incur network traffic.

```
// Pseudo Slice
int ice_getHash();
Communicator ice_getCommunicator();
string ice_toString();
Identity ice_getIdentity();
string ice_getAdapterId();
EndpointSeq ice_getEndpoints();
EndpointSelectionType ice_getEndpointSelection();
Context ice_getContext();
string ice_getFacet();
bool ice_isTwoway();
bool ice_isOneway();
bool ice_isBatchOneway();
bool ice_isDatagram();
bool ice_isBatchDatagram();
bool ice_isSecure();
bool ice_isPreferSecure();
Router* ice_getRouter();
Locator* ice_getLocator();
int ice_getLocatorCacheTimeout();
bool ice_isCollocationOptimized();
bool ice_isThreadPerConnection();
onnection ice_getConnection();
Connection ice_getCachedConnection();
bool ice_isConnectionCached();
```

For example:

```
// C++
ObjectPrx obj = communicator()->stringToProxy(
    "Hello:tcp -h 192.168.1.4 -p 10000");
assert(obj->ice_isTwoway());
```

The `ice_isTwoway` method returns true if the proxy uses twoway invocations; false otherwise.

Factory Methods

These methods create a new proxy with the requested configuration.

```
// Pseudo Slice
Object* ice_identity(Identity id);
Object* ice_adapterId(string id);
Object* ice_endpoints(EndpointSeqendpoint s);
Object* ice_endpointSelection(
    EndpointSelectionType t);
Object* ice_context(Context ctx);
Object* ice_defaultContext();
Object* ice_facet(stringfacet);
Object* ice_twoway();
Object* ice_oneway();
Object* ice_batchOneway();
Object* ice_datagram();
Object* ice_batchDatagram();
```

```
Object* ice_secure(bool b);
Object* ice_preferSecure(bool b);
Object* ice_compress(bool b);
Object* ice_timeout(inttimeout);
Object* ice_router(Router* rtr);
Router* ice_getRouter();
Object* ice_locator(Locator* loc);
Object* ice_locatorCacheTimeout(int seconds);
Object* ice_collocationOptimized(bool b);
Object* ice_connectionId(string id);
Object* ice_threadPerConnection(bool b);
Object* ice_connectionCached(bool b);
```

For example:

```
// C++
ObjectPrx obj = communicator()->stringToProxy(
    "Hello:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 12000");
obj = obj->ice_secure(true);
```

Calling `ice_secure(true)` returns a new proxy that will make invocations only via secure endpoints.

Obtaining Proxies

Stringified Proxies

For bootstrapping purposes, proxies are almost always obtained from a stringified proxy (or via proxy properties—see below). As we saw earlier, stringified proxies can be used to create direct or indirect proxies. Direct proxies have a set of associated endpoints. Each endpoint contains a protocol identifier and associated protocol-specific addressing information that specifies how and where the target object can be reached. For example:

```
// C++
HelloPrx hello = HelloPrx::checkedCast(
    communicator()->stringToProxy(
        "Hello:tcp -h 192.168.1.4 -p 10000"));
```

This creates a direct proxy with the object identity `Hello` that can be contacted on host 192.168.1.4 at port 10000 using the TCP protocol.

Direct proxies can have multiple endpoints. For example:

```
// C++
HelloPrx hello = HelloPrx::checkedCast(
    communicator()->stringToProxy(
        "Hello:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000"));
```

This stringified proxy contains two endpoints. The first is a TCP endpoint (as we just saw in the preceding example). The second is an SSL endpoint for the host 192.168.1.4 and port 11000. A proxy with more than one endpoint tells the Ice run time that the target object can be reached at more than one address. The Ice Manual describes how the Ice run time decides which endpoint to use. (I will describe this in more detail in a future article.)

Stringified indirect proxies can specify a well-known proxy, for example:

```
// C++
HelloPrx hello = HelloPrx::checkedCast(
    communicator()->stringToProxy("Hello"));
```

This code creates an indirect proxy with the associated identity `hello`. Alternatively, stringified proxies can specify an adapter identifier:

```
// C++
HelloPrx hello = HelloPrx::checkedCast(
    communicator()->stringToProxy(
        "Hello@HelloAdapter"));
```

This code creates an indirect proxy with the associated identity `Hello` that resides at the object adapter with the identifier `HelloAdapter`.

Stringified proxies can also specify marshaled proxy options. For example, to set the secure mode, you can provide the `-s` option:

```
// C++
HelloPrx hello = HelloPrx::checkedCast(
    communicator()->stringToProxy("Hello -s"));
```

To specify a facet name, use the `-f` option:

```
// C++
HelloPrx hello = HelloPrx::checkedCast(
    communicator()->stringToProxy("Hello -f v2"));
```

Other options are `-t` for twoway invocations (this is the default), `-o` for oneway invocations , `-O` for batch oneway invocations, `-d` for datagram invocations, and `-D` for batch datagram invocations.

```
// C++
HelloPrx hello = HelloPrx::checkedCast(
    communicator()->stringToProxy(
        "Hello -o:tcp -h 192.168.1.4 -p 10000"));
```

Endpoints can also contain additional flags other than `-h` (for the host) and `-p` (for the port). The exact flags depend on the transport. For TCP and SSL, Ice supports `-t` *`timeout`* to set the timeout and `-z` to set protocol compression. For example:

```
// C++
HelloPrx hello = HelloPrx::checkedCast(
    communicator()->stringToProxy(
        "Hello:tcp -h 192.168.1.4 -p 10000 -t
10000 -z"));
```

This sets a ten-second timeout on all invocations to the object via this proxy, and tells the proxy to use protocol compression (if possible).

For UDP, Ice supports `-z` to to set protocol compression and `-e` and `-v` to set the protocol and encoding versions. (See the Ice Manual for more information on why you might want to use these options.)

```
// C++
HelloPrx hello = HelloPrx::checkedCast(
    communicator()->stringToProxy(
    "Hello -d:udp -h 192.168.1.4 -p 10000 -z"));
```

This example configures the `hello` proxy to use datagrams and configures the UDP endpoint to use protocol compression. (Note that with UDP, if the server hosting the Ice object does not support protocol compression, the message will be lost; because UDP is unidirectional, the client has no direct way to find out about this problem.)

Proxy Properties

Proxy properties are an alternate way to create bootstrap proxies. We introduced proxy properties in Ice 3.2 as a more flexible way of externalizing proxies with property settings (instead of hard-coding stringified proxies). Proxy properties are also the only way to control local proxy settings without making API calls. (Stringified proxies cannot be used to control local proxy settings because they only provide options to control marshaled proxy settings.)

```
# config.client
Hello.Proxy=Hello:tcp -h 192.168.1.4 -p 10000
```

```
// C++ using config.client
HelloPrx hello = HelloPrx::checkedCast(
    communicator()->propertyToProxy(
        "Hello.Proxy"));
```

The `propertyToProxy` method in the preceding example looks up the `Hello.Proxy` property to determine the stringified proxy to use. Of course, you could also do this yourself as follows:

```
// C++ using config.client
HelloPrx hello = HelloPrx::checkedCast(
  communicator()->stringToProxy(
    communicator()->getProperties()->getProperty(
        "Hello.Proxy")));
```

So why are proxy properties useful? Their real advantage is that they allow you to configure local proxy settings. For example:

```
# config.client
Hello.Proxy=Hello:tcp -h 192.168.1.4 -p 10000
Hello.PreferSecure=1
```

This configuration configures the proxy to prefer to secure connections over insecure ones. This is equivalent to writing:

```
// C++ using config.client
ObjectPrx obj = communicator()->stringToProxy(
    communicator()->getProperties()->getProperty(
        "Hello.Proxy"));
obj = obj->ice_preferSecure(true);
HelloPrx hello = HelloPrx::checkedCast(obj);
```

You can also control other local proxy settings via proxy properties, namely, collocation optimization, connection caching, endpoint selection, the locator proxy, the locator cache timeout, the router proxy, and the thread-per-connection concurrency model. (See the Ice Manual for details.)

Proxy Factory Methods

Proxy factory methods provide another way to create new proxies. In all cases, the new proxy is a copy of the original proxy, but with one setting altered. For example, the `ice_secure` method returns a proxy that will make invocations only via secure endpoints:

```
// C++
ObjectPrx obj = communicator()->stringToProxy(
    "Hello:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000");
obj = obj->ice_secure();
HelloPrx hello = HelloPrx::checkedCast(obj);
```

`checkedCast` and `uncheckedCast` also create new proxies of a specific type. `uncheckedCast` returns a new proxy of the specified type. This function is not type-safe in that there is no guarantee that the Ice object to which the proxy refers indeed supports the specified interface. (You must make sure that the type ID you specify for an `uncheckedCast` matches an interface that the target object implements.) If you want to find out whether the target object implements a particular interface, you could call `ice_isA` and, if the interface is supported, call `uncheckedCast`. This is what a `checkedCast` does: it internally calls `ice_isA` to validate the type, and then returns a new proxy if the type is correct, and `null` otherwise. (If the target object is unreachable, `checkedCast` throws an exception.)

Method Invocations

Ice provides stringified proxies mainly for bootstrapping: normally, proxies are returned by making operation invocations but, to make an invocation, the client needs a proxy. Stringified proxies solve this chicken-and-egg problem and allow you to configure clients with the few initial proxies they typically need to "get off the ground". However, once a client has the first few proxies, it should not use stringified proxies any longer and, instead, obtain further proxies may making operation invocations. For example:

```
// Slice
interface Widget
{
     // ...
};
interface WidgetFactory
{
     Widget* create();
};
```

```
// C++
WidgetFactoryPrx factory =
    WidgetFactory::checkedCast(
        communicator->stringToProxy(
            "WidgetFactory")):
WidgetPrx widget = factory->create();
```

Note that this code obtains the proxy to the new widget directly as a value, and no conversion from a string to a proxy and no down-cast are necessary.

In contrast, consider the following:

```
// Slice
interface WidgetFactory
{
     string create();
};
```

```
// C++
WidgetFactoryPrx factory = ...;
// Bad!
WidgetPrx widget = WidgetPrx::uncheckedCast(
    communicator->stringToProxy(
        factory->create()));
```

This is a bad idea. Do not pass stringified proxies over the wire; instead, pass them as proxies. Firstly, passing proxies as strings is less efficient because proxies in marshaled form are more compact than in string form. Secondly, passing proxies as strings bypasses the Slice type system and the guarantees provided by the Slice contract. By passing a proxy as a string the receiver needs to first convert the string back to a proxy of the appropriate type before use, instead of using the type information implicit in the Slice contract. This can result in violations of the type system at run time that, otherwise, would be caught at compile time. (See this FAQ for more details on this topic.)

Fixed and Routed Proxies

As previously stated, fixed proxies can neither be created directly from a stringified proxy, nor obtained as the result of a method invocation. The only way to create a fixed proxy is by calling `createProxy` on a connection object. Doing so creates a fixed proxy that is bound to the corresponding connection.

Routed proxies are created by either setting the `Ice.Default.Router` property or by creating a new proxy from an existing one by calling `ice_router`. A routed proxy sends all invocations on the proxy not to the actual target object, but instead to a router object that, in turn, forwards the invocation to the real target.

Proxy Defaults and Overrides

Ice supports both proxy defaults and proxy overrides, which allow you to control specific property settings even if they are not explicitly set during proxy creation.

Proxy Defaults

A proxy default controls a setting of a proxy if that setting is not explicitly specified. For example:

```
# config.client
Ice.Default.CollocationOptimization=0
```

```
// C++
ObjectPrx obj = communicator()->stringToProxy(
    "Hello:tcp -h 192.168.1.4 -p 10000");
assert(!obj->isCollocationOptimized());
```

By default, all proxies support collocation optimization. (Collocation optimization means that calls on collocated Ice objects take an optimized code path that avoids marshaling and network overhead.) Under some circumstances, collocation optimization is not desirable. (See the Ice Manual for a list of differences between collocated and regular method invocations.) To change this setting for an individual proxy, you can call `ice_collocationOptimized` to explicitly control the setting. However, you can change the default setting, making it unnecessary to disable collocation optimization every time you create a proxy (as shown in the preceding example.)

Let's look at another example:

```
# config.client
Ice.Default.Host=192.168.1.4
```

```
// C++
ObjectPrx obj = communicator()->stringToProxy(
    "Hello:tcp -p 10000");
obj->ice_ping();
```

In this case, since no host is specified for the TCP endpoint, the Ice run time uses the `Ice.Default.Host` property to set the host for the proxy and will contact the host at 192.168.1.4 when the code calls `ice_ping` on the proxy.

Contrast this with the following:

```
# config.client
Ice.Default.Host=192.168.1.4
```

```
// C++
ObjectPrx obj = communicator()->stringToProxy(
    "Hello:tcp -h remote.host.com -p 10000");
obj->ice_ping();
```

Even though `Ice.Default.Host` is set, the proxy still contains the host remote.host.com, because it was explicitly set during proxy creation (so the default is simply ignored).

The defaults that you will use most often in your applications are `Ice.Default.Router` (when using Glacier2) and `Ice.Default.Locator` (when using IceGrid). See the Ice Manual for more default settings that you may find useful in your applications.

Proxy Overrides

In contrast to default settings, override settings are used regardless of any explicit setting. For example:

```
# config.client
Ice.Override.Compress=1
```

```
// C++
HelloPrx hello = HelloPrx::uncheckedCast(
    communicator()->stringToProxy(
        "Hello:tcp -h 192.168.1.4 -p 10000"));
hello->sayHello();
```

In this case, all communications via this proxy always use protocol compression, even though the proxy is created without the `-z` option. (If the server does not support protocol compression, the client receive a `ConnectionLostException`.)

When using overrides, the proxy's local settings are always ignored. For example, `Ice.Override.Secure` instructs the Ice run time to only bind to secure endpoints:

```
# config.client
Ice.Override.Secure=1
```

```
// C++
HelloPrx hello = HelloPrx::uncheckedCast(
    communicator()->stringToProxy(
        "Hello:tcp -h 192.168.1.4 -p 10000"));
hello->sayHello();
```

This invocation throws a `NoEndpointException` because the proxy does not contain a secure endpoint. Consider this example:

```
# config.client
Ice.Override.Secure=1
```

```
// C++
HelloPrx hello = HelloPrx::uncheckedCast(
    communicator()->stringToProxy(
        "Hello:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000"));
hello->sayHello();
```

In this case, invocations are always secure and the TCP endpoint is simply ignored. In contrast, if `Ice.Override.Secure` is not set, insecure endpoints are preferred over secure endpoints (unless `Ice.Default.PreferSecure` is set or `ice_preferSecure(true)` has been called on the proxy in which case secure endpoints are preferred over insecure endpoints).

```
# config.client
Ice.Override.Secure=1
```

```
// C++
ObjectPrx obj = communicator()->stringToProxy(
    "Hello:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000");
obj = obj->ice_secure(false);
HelloPrx hello = HelloPrx::uncheckedCast(obj);
hello->sayHello();
```

In this case, secure communications will still be used, despite the call to `ice_secure(false)`.

Note that overrides do not change the proxy, they only change the behavior of the proxy when it is used. Consider the following two examples, run without `Ice.Override.Secure` being set. Here is the first example:

```
// C++
ObjectPrx obj = communicator()->stringToProxy(
    "Hello:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000");
cout << obj->ice_toString() << endl;
```

```
$ test
Hello -t:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000
```

As you would expect, the stringified proxy (apart from the added `-t` option) is identical to the string that the code passes to `stringToProxy`.

The second example also runs without `Ice.Override.Secure` being set, but calls `ice_secure` explicitly:

```
// C++
ObjectPrx obj = communicator()->stringToProxy(
    "Hello:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000");
obj = obj->ice_secure(true);
cout << obj->ice_toString() << endl;
```

```
$ test
Hello -s -t:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000
```

Note that the stringified proxy now contains a `-s` flag, which indicates that the proxy is secure and will only make invocations over secure endpoints. (This is not surprising, given that the code called `ice_secure(true)` to create the proxy.)

Now consider the first example once more, but run with `Ice.Override.Secure` set:

```
# config.client
Ice.Override.Secure=1
```

```
// C++
ObjectPrx obj = communicator()->stringToProxy(
    "Hello:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000");
cout << obj->ice_toString() << endl;
```

```
$ test
Hello -t:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000
```

Although the proxy acts securely (meaning that it will only make invocations over secure endpoints), when the code calls `ice_toString`, the resulting string does *not* have the `-s` secure flag set. In other words, proxy overrides affect the behavior of a proxy, but do not change a proxy's contents. This distinction is important and you should keep it in mind.

Proxy overrides do not affect proxy comparison. Consider:

```
# config.client
Ice.Override.Secure=1

// C++
ObjectPrx o1 = communicator()->stringToProxy(
    "Hello:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000");
ObjectPrx o2 = communicator()->stringToProxy(
    "Hello -s:tcp -h 192.168.1.4 -p 10000:ssl -h
192.168.1.4 -p 11000");
assert(o1 == o2);
```

This assertion will fail: although the two proxies *behave* the same way at run time due to the `Ice.Override.Secure=1` override, they are not the same.

Proxies are First-Class Types

We occasionally see Ice applications that avoid storing or passing proxies for fear that proxies are heavy-weight objects. Such fears are unfounded: proxies have a very efficient internal representation and attempts to avoid using proxies are likely to result in more CPU and memory overhead, not less. For example:

```
// C++
ObjectPrx o1 = ...;
ObjectPrx o2 = o1;
```

In this case `o1` and `o2` are smart pointers to the same proxy object, and smart pointers provide very efficient initialization and assignment. Contrast this to:

```
// C++
ObjectPrx o1 = ...;
ObjectPrx o2 = communicator->stringToProxy(
    o2->ice_toString());
```

In this case, `o1` and `o2` point to different internal proxy objects, that is, the code consumes memory for two proxy instances, instead of a single one.

The following example illustrates that Ice can avoid creating a new proxy in some cases:

```
// C++
ObjectPrx o1 = communicator->stringToProxy(
    "hello -t 5000");
ObjectPrx o2 = o1->ice_timeout(5000);
```

`o1` and `o2` still point to the same proxy object. Ice is smart enough to realize that the new proxy being created has the same timeout as the source proxy; in this case, it avoids creating a new proxy and simply returns a smart pointer to the already existing proxy. (Because proxies are immutable, this optimization is safe.)

The Ice run time also avoids expensive operations until they become necessary. For example, Ice does not establish a connection once a proxy is created, but only once a connection is actually required in order to invoke an operation (such as `checkedCast`, `ice_isA`, `ice_ping`, `ice_id`, `ice_ids`, or a Slice-defined operation).

Occasionally, designers try to avoid passing proxies as parameters, usually to the detriment of the entire system: the likely outcome is poor performance, inconvenient interfaces, and lack of scalability. Consider this example of incorrect design:

```
// Slice--AWKWARD!
interface Widget
{
     int id();
     // ...
};
exception WidgetExistsException
{
};
interface WidgetFactory
{
     Widget* create(int id)
        raises WidgetExistsException;
     Widget* find(int id);
};
interface WidgetContainer
{
     void store(int id);
     StringSeq getWidgets();
};
```

This application requires a container of widgets. Widgets are created using the factory, and then placed into their container as a widget ID. The factory allows widgets to be located by ID with the `find` operation.

This design harbors problems that are not obvious until you start to use, evolve, and scale the application. For one, the design is awkward and performs poorly: to use a widget that is retrieved from the container, the caller must first re-obtain a proxy to the widget by calling the `find` operation on the factory. This step is not only unnecessary, but also expensive because it requires an additional remote invocation. Second, to look up a widget via its ID, the caller must know *which* widget factory to use (which would be unnecessary if the caller would have proxy to the widget in the first place). You might dismiss this as academic: surely there will be only one widget factory. However, scalability dictates otherwise. Chances are that a once-small application will become a much larger application and, before you know, the application will need multiple widget factories. In turn, to support multiple factories, the original design needs to be modified to provide callers with a means to locate the factory for a given widget ID (unless callers would try all factories, which would be inefficient). Furthermore, extending the original design to multiple factories also requires a scheme to partition the widget IDs such that IDs remain unique across different factories.

The root problem of the design is that, by storing only the ID of widget, location information is lost. A superior design is as follows:

```
// Slice
interface Widget
{
     // ...
};
sequence<Widget*> WidgetPrxSeq;
interface WidgetFactory
{
     Widget* create();
};
interface WidgetContainer
{
     void store(Widget* w);
     WidgetPrxSeq getWidgets();
};
```

This interface exhibits none of the preceding problems. It is straightforward, extensible, easy to use, and performs well. We strongly encourage you to use proxies as they were intended to be used, namely, as strongly-typed values that can be exchanged as easily and efficiently as a string. Doing so results in better performance and does not compromise the type safety of an application.

Master–Slave Replication with Ice

Benoit Foucher, Senior Software Engineer

Introduction

Ice and IceGrid provide facilities that allow you to replicate services and allow clients to transparently use these replicated services. This replication provides load balancing over multiple machines and fault tolerance for vital services.

Clients have several ways to take advantage of replication in Ice:

- **Direct proxies**. A direct proxy can have multiple endpoints and can point to multiple replicas of the same service. For example, you can use the proxy `hello:tcp -p 12345 -h host1.foo.com:tcp -p 12345 -h host2.foo.com` to invoke on the replicated `hello` object. By default, the Ice run time will randomly select one of the two endpoints when a client makes an invocation. If one replica fails, the Ice run time automatically tries the endpoint of the other replica.
- **Indirect proxies with replica groups**. An indirect proxy can point to a replica group, such as `hello@MyReplicaGroup`. The replica group identifier `MyReplicaGroup` is resolved by the Ice locator service to the endpoints of one or more replicas.

So, to use replication, all you need to do is implement a service and deploy multiple instances of it, and distribute the appropriate proxies to clients. Is it really that easy? Well, yes, or at least, it would be if the service were completely stateless. However, very often, services have state.

In this article, I outline how to replicate a simple stateful service using master–slave replication. The service stores information about users in a database and allows clients to query that information. Updates to the database can be made only by the master instance, and slaves can only read, but not update, the database. To keep slaves up to date, the master replicates the database contents to the slaves. If the master goes down, clients can no longer make any updates but can still query the database via the slaves.

The Slice Interfaces

Let's take a look at the Slice interfaces for the service. The service is provided by two interfaces; an interface to modify the user database, and an interface to query the database:

```
// Slice
module Demo
{
exception UserNotFoundException
{
    string id;
};

struct UserInfo
{
    string id;
    string firstName;
    string lastName;
    string address;
};

interface UserQuery
{
    UserInfo get(string id)
        throws UserNotFoundException;
    UserDatabase* getDatabase();
};


interface UserDatabase
{
    void add(UserInfo info);
    void remove(string id);

};
};
```

The user information is stored in a Freeze map. The `add` and `remove` operations of the `UserDatabase` interface allow clients to add and remove users, and the `get` operation of the `UserQuery` interface allows clients to retrieve users via their identity. The `getDatabase` operation returns the proxy of the `UserDatabase` object.

By using separate interfaces, we clearly split the functionality provided by the master from the functionality provided by the slaves. This makes it easy to write slaves such that they provide access only to slave functionality. In contrast, if we had defined a single interface, the master would provide a fully functional implementation, whereas the slaves would provide only a partial implementation. In turn, this would mean that either the slave implementation of the `add` and `remove` methods would need to throw an exception to indicate that these operations are supported only on the master (which is ugly), or the slaves would need to forward `add` and `remove` operations to the master (which is inefficient). For these reasons, it is better to use two separate interfaces.

Implementation without Replication

The implementation of these two interfaces is trivial if we omit replication for now. Here is the bulk of the code:

```
// Java
public class DatabaseI implements
    _UserDatabaseOperations, _UserQueryOperations
{
    synchronized public UserInfo
    get(String id, Ice.Current current)
        throws UserNotFoundException
    {
        UserInfo info =
            (UserInfo)_users.get(id);
        if(info == null)
        {
            throw new UserNotFoundException(id);
        }
        return info;
    }

    synchronized public UserDatabasePrx
    getDatabase(Ice.Current current)
    {
        return _database;
    }

    synchronized public void
    add(final UserInfo info, Ice.Current current)
    {
        _users.fastPut(info.id, info);
    }

    synchronized public void
    remove(final String id, Ice.Current current)
    {
        _users.fastRemove(id);
    }

    DatabaseI(Ice.Communicator communicator,
        UserDatabasePrx database)
    {
        _database = database;
        _connection =
            Freeze.Util.createConnection(
                communicator, "Master");
        _users = new StringUserInfoDict(
            _connection, "users");
    }

    final private UserDatabasePrx _database;
    final private Freeze.Connection _connection;
    final private StringUserInfoDict _users;
}
```

We create a Freeze map and use it to store and retrieve the user information. You might notice that the `DatabaseI` class inherits from the generated `_UserDatabaseOperations` and `_UserQueryOperations`. We're using tie classes and delegation to implement the interfaces. This allows implementing the two interfaces with a single Java class. Here is how the servants are created and registered:

```
// Java
public class Server extends Ice.Application
{
    public int
    run(String[] args)
    {
        Ice.ObjectAdapter adapter =
        communicator().createObjectAdapter(
            "Server");
        UserDatabasePrx proxy =
            UserDatabasePrxHelper.uncheckedCast(
                adapter.createProxy(
                  communicator().stringToIdentity(
                    "UserDatabase")));
        DatabaseI db = new DatabaseI(
            communicator(), proxy);
        adapter.add(new _UserDatabaseTie(db),
            proxy.ice_getIdentity());
        adapter.add(new _UserQueryTie(db),
            communicator().stringToIdentity(
            "UserQuery"));
        adapter.activate();
        communicator().waitForShutdown();
        return 0;
    }
    …
}
```

The generated class `_UserDatabaseTie` and `_UserQueryTie` delegate the implementation of the interfaces to the `DatabaseI` class. The server incarnates two objects whose identities are `UserDatabase` and `UserQuery`. To access the `UserDatabase` and `UserQuery` interfaces, a client needs to be configured with only the proxy of the `UserQuery` interface. It can invoke the `getDatabase` method to retrieve the proxy of the `UserDatabase` interface.

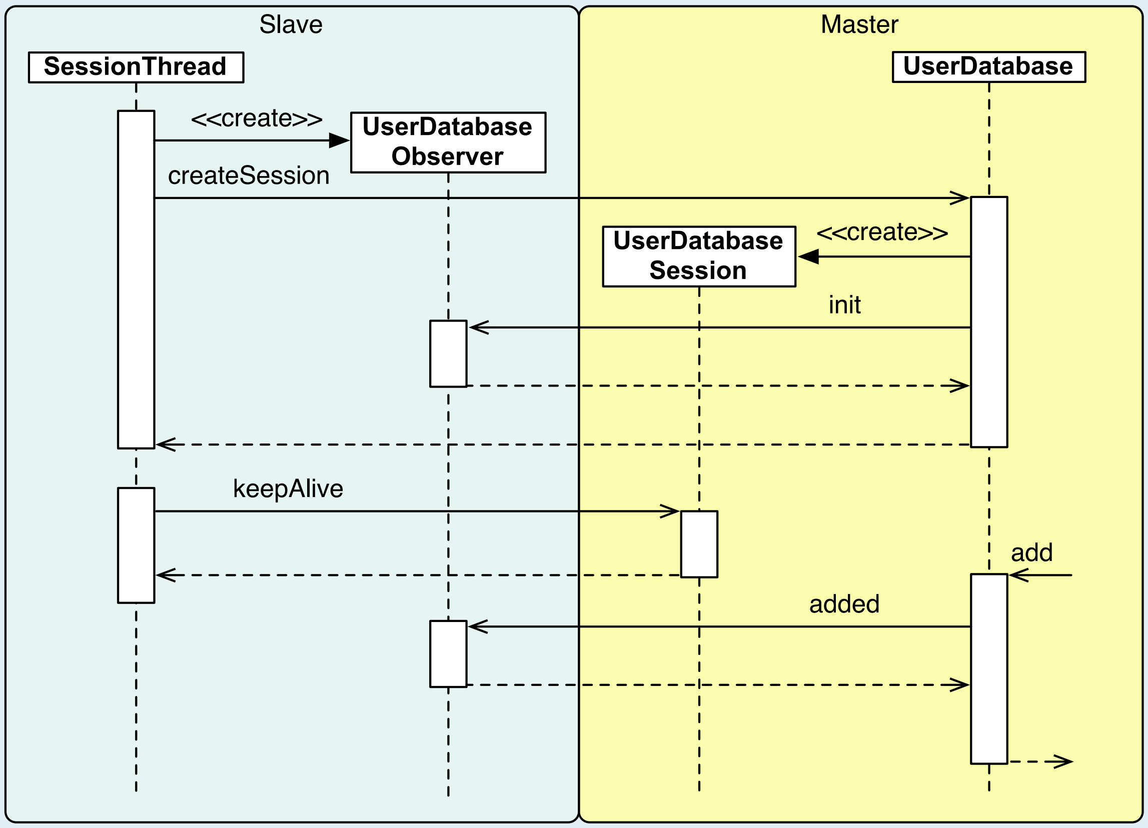
The Slice Interfaces for Replication

Now let's add master–slave replication to this service. To do that, we will use the observer pattern, plus a session between each slave and the master. (See "The Grim Reaper" in Issue 3 of *Connections* for more information on sessions.)

Slaves are observers of the master user database: each time the database is updated, the master sends notifications to its slaves. Each slave maintains an internal database that can be modified only through the observer notifications.

The slaves need to create a session with the master to receive observer updates from the master database. If the master cannot send a notification to a slave (perhaps because of a temporary network problem), the session is destroyed and the slave has to establish a new session to re-synchronize its database with the master database. This ensures the consistency of the slave database with the master database: as long as there is an active session between the master and the slave, the slave has an accurate up-to-date replica of the database. The session makes it easy to keep track of the connection status between the slave and the master.

Figure 1: Interactions between the Master and a Slave



Slaves implement the `UserDatabaseObserver` to receive database updates from the master. The master calls the `init` method once when the session is established by the slave. (See ”IceStorm 3.2” in Issue 21 of *Connections* for a detailed description of a similar observer interface and why this `init` method is necessary.) The master calls the `added` and `removed` methods when a user is added or removed from the database. The `UserDatabaseSession` interface and the `UserDatabase` interfaces are implemented by the master and used by slaves to create sessions. The slave provides a proxy to its observer object when it creates a session. See Figure 1 for a sequence diagram that shows the interactions between the master and the slave.

Replication requires some interactions between the master and slaves. These interactions are defined by three additional Slice interfaces. These interfaces are defined in their own module to clearly separate user interfaces from interfaces used to implement replication. Here are the additional interfaces:

```
module DemoInternal
{
interface UserDatabaseObserver
{
    void init(UserInfoSeq infos);
    void added(UserInfo info);
    void removed(string id);
};

interface UserDatabaseSession
{
    void keepAlive();
    void destroy();
};

interface UserDatabase extends Demo::UserDatabase
{
     UserDatabaseSession* createSession(
        UserDatabaseObserver* obsv);
};
};
```

The slave has a thread that is dedicated to creating the session and keeping it alive. If the master goes down, the thread periodically tries to re-establish the session. Once a session is established, the master calls back on the slave observer to initialize the slave's database. This ensures that both the master and the slave databases are consistent. Once the session is created, the master sends updates to the slave whenever the master database changes.

The implementation with replication

The `UserDatabase` from the `DemoInternal` module interface is implemented by the `MasterDatabaseI` class. This class also implements the `UserQuery` interface. The implementation of the new `createSession` method is shown below:

```
synchronized public UserDatabaseSessionPrx
createSession(UserDatabaseObserverPrx observer,
    Ice.Current current)
{
    try
    {
        observer.init(
            (UserInfo[])_users.values().toArray(
                new UserInfo[0]));
        // Close the iterator implicitly opened
        // by the call to _users.values()
        _users.closeAllIterators();
    }
    catch(Ice.LocalException ex)
    {
         return null;
    }
    UserDatabaseSessionI session =
        new UserDatabaseSessionI(
            observer, current.adapter);
    _sessions.add(session);
    return session.getProxy();
}
```

First, `createSession` invokes the `init` method on the observer. This invocation provides the list of all the users currently stored in the master database to the observer and allows the slave to synchronize its database with the master database. If the `init` call is successful, `createSession` creates a new session servant and adds it to the `_slaves` list. The `MasterDatabaseI` class uses this list to keep track of the active sessions and their observer proxies.

In addition to modifying the database, the implementations of the `add` and `remove` need to notify the observers. The `notifyObservers` method achieves this with an appropriate `Update` object. For example, here is the `add` operation:

```
synchronized public void
add(final UserInfo info, Ice.Current current)
{
    _users.fastPut(info.id, info);
    notifyObservers(new Update()
        {
            public void
            invoke(UserDatabaseObserverPrx proxy)
            {
                proxy.added(info);
            }
    });
}
```

The implementation of the `notifyObservers` method is shown below:

```
private void
notifyObservers(Update update)
{
    java.util.Iterator p = _sessions.iterator();
    while(p.hasNext())
    {
        UserDatabaseSessionI session =
            (UserDatabaseSessionI)p.next();
        try
        {
            DemoInternal.UserDatabaseObserverPrx
                observer = session.getObserver();
            if(observer != null)
            {
                update.invoke(observer);
            }
            else
            {
                // The session has been destroyed
                // by the slave.
                p.remove();
            }
        }
        catch(Ice.LocalException ex)
        {
            _logger.warning(
                "lost connection with replica:\n"
                 + ex.toString());
            session.destroy(null);
            p.remove();
        }
    }
}
```

Each time a user is added or removed, the `add` or `remove` method calls `notifyObservers` to notify the observers. The `getObserver` method of the `UserDatabaseSessionI` class returns the proxy of the observer associated with the session. If the session is destroyed, `getObserver` returns `null` and `notifyObservers` removes the session from the list of sessions. Otherwise, `notifyObservers` calls `invoke` on the update object, which in turn calls the observer to send the appropriate update. If the invocation fails, `notifyObservers` destroys the session and removes it from the `_sessions` list of sessions. The `notifyObservers` method really has two functions:

- sending updates to all the observers with an active session,
- reaping destroyed sessions.

Destroying the session if an update fails ensures that the slave will re-connect, thereby creating a new session with the master. By creating a new session, the slave again receives all the users currently stored in the database, which ensures that the slave database is synchronized with the master.

The slave implements the `UserDatabaseObserver` and `UserQuery` interfaces in the `SlaveDatabaseI` class. Similar to the master, this class stores the users in a Freeze map, to allow upgrading a slave to a master by re-using its database. The implementation of the observer interface is trivial and shown below:

```
synchronized public void
init(UserInfo[] infos, Ice.Current current)
{
    _users.clear();
    for(int i = 0; i < infos.length; ++i)
    {
        UserInfo info = infos[i];
        _users.fastPut(info.id, info);
    }
}

synchronized public void
added(UserInfo info, Ice.Current current)
{
    _users.fastPut(info.id, info);
}
```

```
synchronized public void
removed(final String id, Ice.Current current)
{
    _users.fastRemove(id);
}
```

The implementation of the `init` method simply replaces the contents of the slave database with the users provided by the master. The implementation of the `added` and `removed` methods adds and removes users from the slave database.

The master sends updates with twoway synchronous calls from the synchronization block of the `MasterDatabaseI` servant methods. This is necessary to ensure that updates are not sent out of order. For instance, the `init` call on the observer must be made before any `added` or `removed` call on the observer.

Finally, the slave must establish a session with the master to start receiving updates. This is achieved with a dedicated keep-alive thread that has two roles:

- Create the session if it is not established already. The thread tries to create the session at regular intervals until it succeeds.
- Keep the session alive if it is established already. The thread tries to recreate a session if it detects that the session is unreachable or has been destroyed.

The keep-alive thread also creates the `UserDatabaseObserver` servant for each session and registers it with the object adapter. It is important to use distinct observer objects for each session, otherwise the slave could receive updates from a previous observer (if the master has not yet detected that the previous session was destroyed by the slave).

Caveats and Optimizations

The implementation of the master–slave replication ends up being quite simple thanks to the use of a session and the observer design pattern. However, there are a few caveats and optimizations you need to be aware of.

Consistency

Since the replication updates are propagated synchronously by the master to the slaves, our replicated user database provides "read your writes" consistency to clients. In other words, if a client updates the database, it is guaranteed to be able to read the modification once the update request has completed. A client is also guaranteed to always retrieve sequential updates if it uses the same slave to read the data.

However, inconsistencies among the slaves are possible because observer updates are not sent atomically. It is possible for an update to be visible on a slave but not visible on another slave yet (because of network latency or because the slave is currently not connected to the master). So, if a client uses multiple slaves to read the data, it might not get a sequential view of the database updates.

The application has to be aware of this limitation and, if necessary, deal with it. One simple solution is to make sure that the application always uses the same slave to read updates. Another solution is to improve the replication mechanism to ensure that all slaves have the same view of the master database at the same point in time. (However, this requires a more elaborate protocol to distribute the updates, similar to the two-phase commit protocol used for distributed transactions.)

Observer Updates

Observer updates are sent with twoway calls within the master servant synchronization block. This is necessary for two reasons:

- Updates to observers need to be sent within the synchronization to make sure the database updates and the observer updates are atomic. Without synchronization, we would get inconsistencies if two threads concurrently modify the same database entry. For example, a thread could add user `u` to the Freeze map and be interrupted by the operating system scheduler. Another thread could then remove `u` from the Freeze map and notify the observer that user `u` was removed. When the first thread is scheduled again, it would then notify the observer that `u` was added, resulting in an inconsistent slave database.
- The updates need to be sent with twoway calls to ensure that the updates are received in the correct order. (Oneway cannot be used because they may be dispatched out of order, depending on the observer configuration.)

The problem with synchronized twoway calls is that, if an observer invocation takes a while to complete, it locks up the master database, and invocations from clients to add or remove users hang until the observer invocation completes.

One way to solve this issue is to set a timeout on the observer proxies that limits the amount of time the lock is held for. Another solution is to delegate the sending of observer updates to a separate thread. The updates can be queued within the synchronization block of the master servant and picked up by a sender thread to be sent in-order using twoway calls. Finally, another option is to use IceStorm to distribute the observer updates. However, because IceStorm communications are unidirectional, this requires an acknowledgment mechanism to ensure that all updates are correctly delivered to slaves. (This is necessary if IceStorm cannot deliver an update to an observer, in which case it automatically unsubscribes the observer without the master being aware of this.)

Slave Synchronization

The synchronization of the slave with the master consists of simply sending the content of the user database to the slave when the session is established. This is simple, but does not scale to large data sets.

For one, if there are many users, sending the list of users in a single invocation could throw a `MemoryLimitException`. (A better approach would be to send the database in multiple invocations, for example, with batch oneway invocations.)

But, even assuming that we can send the updates without running into memory limitations, the approach still does not scale beyond a certain point. In that case, it is better have the master store a log of all the changes made to the user database. The synchronization of a slave with the master then consists of replaying all the changes that have occurred to the master database since the last time the slave was connected. This requires the slave to keep track of a log identifier to identify the last update it received from the master, and it requires policies to determine when the log can be cleaned up because it is impossible to indefinitely track all of the changes made to the user database. (Of course, if the slave database is empty, this still requires transfer of the whole database, but only once, instead of every time a session is established.)

Improved Start-Up

While the master is down, some slaves might have an out-of-date image of the master database. When the master becomes available again, it can take some time for the slaves to re-connect and re-synchronize their database with the master database. To get the slaves up-to-date as quickly as possible, the master can notify its slaves whenever it starts up, thereby avoiding this delay.

Master Upgrade

Because the slave and the master use the same database format, it is possible to upgrade a slave to a master by terminating a slave and restarting the slave as a master. However is better if the same thing can be achieved without having to restart the slave process. The main hurdle here is to figure out a way to inform other slaves of the new master without having to edit each slave's configuration file.

One simple way to solve this is to use the Ice locator mechanism. For example, you can register the `UserDatabase` object as a well-known object. When a slave is upgraded to a master, you can update the endpoints of the well-known object in the location service, so no slave configuration needs to change.

Conclusion

This concludes my introduction to the implementation of master–slave replication with Ice. As always, the implementation of something you might have thought quite complicated is very simple with Ice. A few simple interfaces are sufficient for the interactions between the master and slaves.

Beware however, that replication isn't that simple! As we saw, the example implementation presented here has some problems and, if you intend to replicate a large database, you are probably better off looking at a database that provides built-in replication (such as BerkeleyDB).

You can find the source code for this article in the `replication` directory of the archive for this issue. Please get in touch with us in our developer forums if you have any questions or comments.

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/forums/ and deal with specific problems that developers tend to encounter, and for which the answer may not be readily apparent from reading the documentation. We hope that you will find the hints and explanations in this section useful.

Q: How can I `fork` and `exec` an Ice process?

Under Unix, if you have an Ice process (client or server) and want to create a new process, you will have to call `fork` and `exec`. The basic code pattern looks something like the following:

```
// C++
pid_t pid = fork();
switch(pid)
{
    case -1:
    {
        throw "cannot fork";
    }
    case 0:
    {
        // Child

        // ...

        const char* exe = ...;
        const char** argv = ...;
        execv(exe, argv);
    }
    default:
    {
        // Parent
    }
}
```

This looks harmless enough but, unless you do things right, chances are that your process might hang, crash, or do something else unexpected. Here are a few simple rules to make sure things work as intended.

1. Close open file descriptors before calling `exec`.
2. Only call async-signal-safe system calls in the child.
3. Do not call Ice-related functions in the child.
4. If the parent uses asynchronous signal handlers, disable signal delivery before calling `fork`.
5. If the parent uses `Ice::Application` or `IceUtil::CtrlCHandler`, and the child process needs the default behavior for `SIGHUP`, `SIGINT`, and `SIGTERM`, reset these signals to their default behavior in the child before calling `exec`.
6. If `exec` fails, call `_exit`.

Closing open file descriptors in the child is important because not doing so wastes kernel resources and can also interfere with the parent (for example, prevent connection closure when the parent closes a socket that is held open by the child).

Once `fork` has succeeded, the code must only call async-signal-safe system calls. (The Unix attributes(5) man page provides a list of these system calls.) Making any other system call can potentially crash the child.

You must not call Ice-related APIs in the child before calling `exec`. To understand why this is necessary, consider how `fork` works for a threaded process. In essence, `fork` duplicates the entire virtual memory image of the parent and arranges for `fork` to return zero in the child process. In addition, if the parent is threaded, the parent threads are not cloned in the child; instead, `fork` creates a single thread in the child (which is the thread that returns from the call). However, because the child has a memory image that is identical to that of the parent, any thread-related data structures will simply be in the state they were in when the parent called `fork` and the kernel made a snapshot of the parent's memory. Among other things, this means that mutexes may remain locked in the child, and data structures may be in an inconsistent state because other threads may have been inside a critical region at the time the parent called `fork`.

If you call any Ice-related function in the child before calling `exec`, things can go badly wrong because the function may attempt to lock a mutex that was already locked at the time the parent called `fork`, causing the child to deadlock. Similarly, the function might call a library function that is not async-signal-safe, causing the child to crash.

If your application installs signal handlers, you need to take extra care. After a `fork`, the child process has the same signal disposition as the parent: signals that are caught and handled by the parent are also caught and handled by the child. It is possible that a signal is delivered to the child before the child can call `exec`. In this case, if the parent handles the signal, so will the child. Depending on what the signal handler does, things can go badly wrong. For one, the signal handler cannot make system calls that are not async-signal-safe—doing so can crash either parent or child. But, even if the signal handler is async-signal-safe, it may have side-effects that are detrimental if the signal arrives in the child before the `exec`. If so, you need to block signal delivery before the parent calls `fork`, and unblock it again in the parent after `fork` returns.

If you use the `Ice::Application` or the `IceUtil::CtrlCHandler` helper classes to handle signals, there is no problem. The Ice run time does not install any signal handlers. Instead, the helper classes block delivery of `SIGHUP`, `SIGINT`, and

SIGTERM and use a dedicated thread that calls sigwait to synchronously accept signals. In turn, this means that your callback functions (set with Application::callbackOnInterrupt or CtrlCHandler::setCallback) can safely call into the Ice run time, and can safely call functions that are not async-signal-safe. However, if you do use these helper classes and call fork, the child process will block SIGHUP, SIGINT, and SIGTERM. If you need the default behavior for these signals in the child, you need to unblock them before calling exec. (See this FAQ for more information.)

Finally, if the exec fails for any reason, you must call _exit (*not* exit). The difference between the two calls is that _exit terminates the process immediately and does not perform any clean-up actions (such as calling atexit handlers). In turn, this means that the destructors of C++ global and static objects are *not* called when you call _exit (whereas, if you call exit, they *are* called). Preventing destructors from running if exec fails is important because, if destructors were to run, they could fail because of the same inconsistent data structures that may be encountered by a signal handler. (Ice uses a few global objects internally, so this rule applies even if you do not have any global objects in your own code.)

So, here is an outline of the code needed to correctly fork and exec:

```
// C++
// Set up a pipe so the child can report errors.
int fds[2];
if(pipe(fds) == -1)
{
    throw "cannot create pipe";
}

// Set close-on-exec on write end of pipe.
int flags = fcntl(fds[1], F_GETFD);
if(flags == -1)
{
    throw "cannot get fcntl flags";
}
flags |= FD_CLOEXEC;
if(fcntl(fds[1], F_SETFD, flags) == -1)
{
    throw "cannot set close-on-exec";
}

// If the parent uses signal handlers,
// block signal delivery here.

pid_t pid = fork();
switch(pid)
{
    case -1:
    {
        throw "cannot fork";
    }
    case 0:
    {
        // Child

        // If the parent uses Ice::Application or
        // IceUtil::CtrlCHandler, and the child
        // requires the default behavior for
        // SIGHUP, SIGINT, and SIGTERM, reset
        // these signals to the default behavior
        // here.

        // Close all open file descriptors.
        int maxFd = static_cast<int>(
            sysconf(_SC_OPEN_MAX));
        for(int fd = 0; fd < maxFd; ++fd)
        {
            // Don't close write end of pipe.
            if(fd != fds[1])
            {
                close(fd);
            }
        }

        const char* exe = ...;
        char* const argv[] = ...;
        execv(exe, argv);

        const char msg[] = "exec failed";
        write(fds[1], msg, sizeof(msg) - 1);
        _exit(1);
    }
    default:
    {
        // Parent

        // Close the write end of the pipe.
        close(fds[1]);

        // Wait for child to write error message
        // or exec successfully.
        stringstream err;
        char c;
        while(read(fds[0], &c, 1) > 0)
        {
            err << c;
        }
        close(fds[0]);
        string msg = err.str();

        // If the parent uses signal handlers,
        // restore signal delivery here.

        if(!msg.empty())
        {
            throw msg;
        }
    }
}
```

Note that this code will most likely need fleshing out for your application. For example, it simply closes all file descriptors, including stdin, stdout, and stderr. It is likely that you will instead want to connect these descriptors to a file or terminal or, if you do not need them, re-open them to `/dev/null`. (Leaving the standard file descriptors closed is bad practice because third-party libraries sometimes fail if these descriptors do not work.) You may also want to perform additional actions, such as copying the parent's environment variables for the child, changing the working directory, setting the process group, or similar. For more details on how to do this, you can consult a Unix book such as Advanced Programming in the Unix Environment, which is excellent.

Also note that, if `exec` fails, the preceding code reports the error instead of having the child exit silently. A common way to implement this (and used by the preceding code) is to call `pipe` before forking to create a pipe between parent and child and to set the close-on-exec flag for the writing end of the pipe. The child writes to the pipe if something goes wrong, and the parent reads the error message from the pipe; the parent's `read` either succeeds and reads the error message or returns with an error if the child called `exec` successfully because, in that case, the kernel closes the writing end of the pipe.

Q: How are connections shared among proxies?

There is a long answer and a short answer to this question. The long answer includes an explanation of how the Ice run time keeps track of connections.

Each proxy stores information about the endpoint(s) at which its object can be reached. When a client invokes an operation on a proxy, the Ice run time checks whether it already has a compatible open connection to the proxy's selected endpoint. If so, it re-uses that connection; otherwise, it establishes a new one. Once established, the run time stores connections in an internal connection table that keeps track of all open connections. That way, a client can use thousands of proxies while using only as many connections as there are distinct endpoints in these proxies. (Note that, if proxies have different timeout values, the Ice run time creates a separate connection, that is, for proxies to objects at the same endpoint, there are as many connections as there are distinct timeouts.)

Connections are closed when, for example, you destroy a communicator and when a request times out or encounters a communication failure. You can also explicitly close a connection via a proxy's associated `Connection` object. If you have automatic connection management (ACM) enabled, the Ice run time periodically closes connections that have been idle for some time, so connections are not held open indefinitely. (See the Ice Manual for more information on `Connection` objects and ACM.)

The short answer as to how connections are shared is "as much as possible"—the Ice run time never opens a connection unless it has to and, if you enable ACM, automatically closes connections when they are no longer needed.