The Fundamentals of Proxies

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

On this page:

Overview of Proxies

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++

```
shared_ptr<HelloPrx> p = ...;
p->sayHello();
```

This example looks remarkably similar to the original code, except that we have used the proxy class `HelloPrx` instead of a pointer to the `Hello` class, and we're using the standard C++ type `std::shared_ptr` to store a smart pointer to an instance of `HelloPrx`. What exactly then is this `HelloPrx` class? An instance of this class refers to an Ice object that supports 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 we saw 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 pass a *stringified* proxy to `stringToProxy` and convert that string into a proxy object:

C++

```
shared_ptr<HelloPrx> p = Ice::checkedCast<HelloPrx>(
    communicator->stringToProxy("hello:tcp -h 192.168.1.4 -p 10000"));
p->sayHello();
```

We will explore 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 requires the proxy to use a secure connection;
- 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 the proxy should prefer secure connections over insecure connections;
- a default `Ice::Context` to use when making invocations;
- a connection ID;
- a timeout for invocations;
- whether to use protocol compression.

Note that proxies are strongly typed in languages that support it, 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 invocation timeout. You can set this timeout by calling `ice_invocationTimeout`:

C++

```
shared_ptr<HelloPrx> proxy = ...;
proxy = proxy->ice_invocationTimeout(5000);
```

The call to `ice_invocationTimeout` creates a new proxy that is identical to the source proxy, except for the new invocation timeout of five seconds. In C++, no down-cast is necessary on the proxy that is returned by *factory methods* such as `ice_invocationTimeout`. Some other language mappings require a regular language cast:

Java Compat

```
HelloPrx proxy = ...;
proxy = (HelloPrx)proxy.ice_invocationTimeout(5000);
```

The cast is required here because `ice_invocationTimeout` 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++

```
shared_ptr<HelloPrx> proxy = ...;
proxy->ice_invocationTimeout(5000); // ERROR!
```

This does not change the invocation timeout on the proxy; instead, it returns a new proxy with a five-second invocation 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++

```
shared_ptr<HelloPrx> h1 = ...:
shared_ptr<HelloPrx> h2 = h1->ice_invocationTimeout(5000); // Set a new timeout
assert(Ice::targetEqualTo(h1, h2)); // Assertion fails
```

In most language mappings, proxies can be compared using the language's standard comparison operators. The C++11 mapping requires a bit more effort to compare `shared_ptr` targets, therefore this mapping provides some additional convenience functions, such as the `targetEqualTo` function demonstrated above.

This assertion fails because the two proxies have different invocation timeout values. Usually, the programmer's actual intent is to discover whether two proxies denote the same Ice object. To do this, you need to compare the object identities:

C++

```
shared_ptr<HelloPrx> h1 = ...;
shared_ptr<HelloPrx> h2 = h1->ice_invocationTimeout(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++

```
shared_ptr<HelloPrx> h1 = ...;
shared_ptr<HelloPrx> h2 = h1->ice_invocationTimeout(5000); // Set a new timeout
assert(proxyIdentityEqual(h1, h2)); // Assertion passes
```

This code is equivalent to the preceding example, which extracted the identities and compared them 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 in other language mappings.)

Slice Proxies

Consider the following C++ code:

C++

```
class Widget { }
class WidgetFactory
{
    Widget* create();
}
```

Compare this to the following Slice:

Slice

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

The C++ `create` method returns a *pointer* to a widget, and the Slice `create` operation returns a *proxy* for a widget. Thus, the Slice syntax `Widget*` means "return a *proxy* to a widget".

Back to Top ^

Proxy Types

Proxies come in several varieties. All of them contain the identity of the associated Ice object and information such as an invocation 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 Teach Yourself IceGrid in 10 Minutes.)

Indirect proxies have two forms. The first is called 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++

```
shared_ptr<HelloPrx> p = Ice::checkedCast<HelloPrx>(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 specified 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++

```
shared_ptr<HelloPrx> p = Ice::uncheckedCast<HelloPrx>(communicator->stringToProxy("hello@HelloAdapter"));
p->sayHello();
```

Note that both direct and indirect proxies may also 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 Teach Yourself Glacier2 in 10 Minutes for more information on Glacier2.)

Fixed Proxies

A fixed proxy is bound to a particular connection for the entire life time of the proxy. Once that connection is closed, the proxy no longer works (and will never work again). In addition, a fixed proxy cannot be marshaled. A server uses a fixed proxy to call back on an object provided by the client without opening a separate outgoing connection from server to client.

Back to Top ^

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.

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

For example:

C++

```
shared_ptr<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, or 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.

```
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();
int ice_getInvocationTimeout();
string ice_getConnectionId();
Connection ice_getConnection();
Connection ice_getCachedConnection();
bool ice_isConnectionCached();
```

For example:

C++

```
shared_ptr<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, or false otherwise.

Factory Methods

Factory methods create a new proxy with the requested configuration.

```
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(int timeout);
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_invocationTimeout(int timeout);
Object* ice_connectionId(string id);
Object* ice_connectionCached(bool b);
```

For example:

C++

```
shared_ptr<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.

Back to Top ^

Obtaining Proxies

Applications have several ways of obtaining proxies.

Stringified Proxies

For bootstrapping purposes, proxies are nearly always obtained from a stringified proxy or a proxy property. 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++

```
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(
    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++

```
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(
    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. Connection Management in Ice describes how the Ice run time decides which endpoint to use.

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

C++

```
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(communicator->stringToProxy("Hello"));
```

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

C++

```
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(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, you can include the `-s` option to enable secure mode:

C++

```
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(communicator->stringToProxy("Hello -s"));
```

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

C++

```
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(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. The example below uses the `-o` option to create a oneway proxy:

C++

```
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(
    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 connection timeout and `-z` to set protocol compression. For example:

C++

```
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(
    communicator->stringToProxy("Hello:tcp -h 192.168.1.4 -p 10000 -t 9000 -z"));
```

This sets a nine-second connection timeout on network activity via this proxy, and tells the proxy to use protocol compression (if possible).

For UDP, Ice supports `-z` 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++

```
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(
    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 bootstrap proxies, providing a more flexible mechanism of externalizing proxies via property settings that avoids the need to hard-code stringified proxies. Proxy properties are also the only way to control local proxy settings without making API calls. (Stringified proxies only provide options for proxy settings that affect the proxy's marshaled state.)

config.client

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

C++

```
// Using config.client
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(communicator->propertyToProxy("Hello.Proxy"));
```

The `propertyToProxy` method in the preceding example interprets the value of the `Hello.Proxy` property as a stringified proxy. Of course, you could also do this yourself as follows:

C++

```
// Using config.client
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(
    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.Proxy.PreferSecure=1
```

The `PreferSecure` property configures the proxy to prefer secure connections over insecure ones. This is equivalent to writing:

C++

```
// Using config.client
shared_ptr<ObjectPrx> obj = communicator->stringToProxy(
    communicator->getProperties()->getProperty("Hello.Proxy"));
obj = obj->ice_preferSecure(true);
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(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, and the router proxy. (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++

```
shared_ptr<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();
shared_ptr<HelloPrx> hello = Ice::checkedCast<HelloPrx>(obj);
```

`checkedCast` and `uncheckedCast` also create new proxies of a specific type. `uncheckedCast` unconditionally returns a new proxy of the requested type. This function is not type-safe in that there is no guarantee that the Ice object to which the proxy refers actually 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.)

Operation 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 by making operation invocations. For example:

Slice

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

C++

```
shared_ptr<WidgetFactoryPrx> factory = Ice::checkedCast<WidgetFactory>(communicator->stringToProxy
("WidgetFactory")):
shared_ptr<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++

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

This is a bad idea. Do not pass stringified proxies over the wire; instead, pass them as proxies. First, passing proxies as strings is less efficient because the marshaled form of a proxy is more compact than its string form. Second, passing proxies as strings bypasses the Slice type system and the guarantees provided by the Slice contract. Passing a proxy as a string forces the receiver to convert the string back to a proxy of the appropriate type before use, rather than 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 How should I pass proxies? 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 its connection and allows the server to make bidirectional requests on the client.

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.

Back to Top ^

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 is used if the application does not explicitly configure a different value for a proxy setting. As an example:

```
Ice.Default.CollocationOptimization=0
```

C++

```
shared_ptr<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 network overhead.) Under some circumstances, collocation optimization is not desirable. To change this setting for an individual proxy, you can call `ice_collocationOptimized` to explicitly control the setting. However, you can also change the default setting (as shown above) so that it is unnecessary to disable collocation optimization every time you create a proxy.

Let's look at another example:

```
Ice.Default.Host=192.168.1.4
```

C++

```
shared_ptr<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:

C++

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

The proxy in this example uses the host `remote.host.com` and ignores the setting of `Ice.Default.Host`.

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

Proxy overrides take precedence regardless of any explicit setting. For example:

```
Ice.Override.Compress=1
```

C++

```
shared_ptr<HelloPrx> hello = Ice::uncheckedCast<HelloPrx>(
    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 receives a `ConnectionLostException`.)

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

```
Ice.Override.Secure=1
```

C++

```
shared_ptr<HelloPrx> hello = Ice::uncheckedCast<HelloPrx>(
    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. Now consider this example:

```
Ice.Override.Secure=1
```

C++

```
shared_ptr<HelloPrx> hello = Ice::uncheckedCast<HelloPrx>(
    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 by default over secure endpoints. (You can change this preference by setting `Ice.Default.PreferSecure` or calling `ice_preferSecure(true)` on the proxy.)

Here is another example:

```
Ice.Override.Secure=1
```

C++

```
shared_ptr<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);
shared_ptr<HelloPrx> hello = Ice::uncheckedCast<HelloPrx>(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 run-time behavior of the proxy. Consider the following two examples, run without `Ice.Override.Secure` being set. Here is the first example:

C++

```
shared_ptr<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;
```

When executed, this code generates:

```
$ 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++

```
shared_ptr<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;
```

Here is the output:

```
$ 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:

```
Ice.Override.Secure=1
```

C++

```
shared_ptr<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;
```

Despite the override setting, the proxy remains unchanged:

```
$ 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.

Proxy overrides also have no effect on proxy comparison. Consider:

```
Ice.Override.Secure=1
```

C++

```
shared_ptr<ObjectPrx> o1 = communicator->stringToProxy(
    "Hello:tcp -h 192.168.1.4 -p 10000:ssl -h 192.168.1.4 -p 11000");
shared_ptr<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); // FAIL!
```

This assertion will fail: although the two proxies *behave* the same way at run time due to the override setting, they are not equivalent.

Back to Top ^

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++

```
shared_ptr<ObjectPrx> o1 = ...;
shared_ptr<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++

```
shared_ptr<ObjectPrx> o1 = ...;
shared_ptr<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 just one.

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

C++

```
shared_ptr<ObjectPrx> o1 = communicator->stringToProxy("hello -o:...");
shared_ptr<ObjectPrx> o2 = o1->ice_oneway();
```

`o1` and `o2` still point to the same proxy object. Ice is smart enough to realize that the source proxy is already a oneway proxy; in this case, it avoids creating a new proxy and simply returns a smart pointer to the 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)
        throws WidgetExistsException;
    Widget* find(int id);
}
interface WidgetContainer
{
    void store(int id);
    IntSeq 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:

1. 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.
2. To look up a widget via its ID, the caller must know *which* widget factory to use (this would be unnecessary if the caller had a proxy for 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 shown below:

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.

Back to Top ^

See Also

- Proxies for Ice Objects
- Proxies
- Proxy and Endpoint Syntax
- Proxy Properties