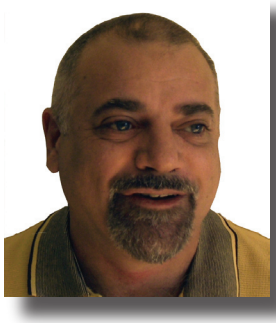
Connections

ZeroC's Newsletter for the Ice Community

Issue Number 15, July 2006



The Emperor has No Clothes

Last month, ACM Queue published "The Rise and Fall of CORBA". It certainly looks like the article touched a nerve with many people, and I received a lot of comments about it. (The overwhelming majority of comments was very positive). Briefly, the article examines why CORBA failed to live up to its promises and concludes that, apart from technical failings, CORBA's demise was caused by the *process* that was used to create it: software consortiums inevitably encourage design-by-committee, incompetence, political infighting, jostling for position, conflict of interest, distrust among the participants, and dishonesty. The end result of this process is something that, at best, is mediocre (and, often, far worse than that).

The worrying thing in all this is not the trouble with CORBA—CORBA is merely one victim. What *really* worries me is that much of the software industry (vendors and customers alike) insists on adhering to this process, even though it pretty much guarantees failure. As an example, Web services, the current silver bullet of distributed computing, uses a process that is much the same, and it seems inevitable that the end result will be very similar. Many readers sent me e-mail about their experiences with other standards bodies, and indicated that they observed many of the same procedural problems there. It is obvious that these problems are not isolated incidents but are endemic to our industry.

We know from bitter experience that too many cooks spoil the broth: Fred Brooks made that abundantly clear in his excellent book *The Mythical Man Month*, first published in 1975. In the 20th anniversary edition of his book, Fred Brooks wrote: *"To my surprise and delight, The Mythical Man-Month continues to be popular after 20 years."* That is quite a stunning remark: after all, how many computing books do you know that are still popular 20 years after they were written? (I can think of only a handful, among them Donald Knuth's brilliant series *The Art of Computer Programming*). In 1999, Ed Yourdon published another book: *Death March*. Yourdon's book is also popular and, significantly, it teaches many of the same lessons that Fred Brooks taught nearly 25 years earlier. It now is the year 2006, and both books are just as popular as ever. I cannot help but conclude that, in more than three decades, the sum total of what the industry has learned about the process of creating software is depressingly close to zero.

Wikipedia provides the following explanation for the title of this editorial:

"Most frequently, the metaphor involves a situation wherein the overwhelming (usually unempowered) majority of observers willingly share in a collective ignorance of an obvious fact, despite individually recognizing the absurdity."

There is an oxymoron in this explanation: "*usually unempowered . . .willingly share . . . despite individually recognizing the absurdity"*. How can people recognize what is happening yet do nothing about it? "Empowerment" means *"to invest with power, especially legal power or official authority"* (dictionary.com). People do not lose their power or authority by accident: they lose it by choice, namely when they choose to remain silent in the face of a ridiculously incongruous process: it is not the situation that is disempowering—it is the act of remaining silent.

For our industry to be taken seriously, and to be trusted and respected, we must stop treating customers like guinea pigs and project strong personal ethics and commitment. As software professionals, we must refuse to participate in development processes that have been known to be ineffective for decades. For the sake of your professional integrity, and for the sake of our industry, the next time you see such a process, I encourage you to speak up instead of joining a column of naked emperors marching toward their grave…

Michi Henning
Chief Scientist

Issue Features

New Features in Ice 3.1/API Changes in Ice 3.1

These two articles discuss the changes in the Ice 3.1 release.

Integrating Ice with a GUI: Part III

The final article in this series presents a technique for updating the UI outside of the main thread.

Contents

New Features in Ice 3.1

With the release of Ice 3.1, we have added quite a number of new features. This article provides an overview of what has changed in IceGrid, IceSSL, and the Java language mapping.

IceGrid Improvements in Ice 3.1

With the 3.1 release, IceGrid includes a number of improvements that help you to better utilize and secure deployed resources, and to more efficiently configure the deployed applications of your grid. The administrative GUI also provides significant improvements.

Allocation Facility

IceGrid allows you to easily deploy services on your network. These services can be discovered by clients with the `IceGrid::Query` interface or by using well-known proxies. However, one thing that was lacking in the previous IceGrid release was a way to coordinate access by clients to services. For example, if a client used an instance of a service to do some computation, there was no easy way to prevent other clients from using the same service.

The new IceGrid allocation mechanism addresses this shortcoming. It allows you to easily coordinate access by clients to Ice objects or servers through the `IceGrid::Session` interface. Clients first need to establish a session with the IceGrid registry to get access to an instance of the `IceGrid::Session` interface. This session can either be established directly with the IceGrid registry through operations on the `IceGrid::Registry` interface, or with Glacier2 by configuring the router to use session managers provided by IceGrid.

The `IceGrid::Session` interface provides two operations to allocate objects, `allocateObjectById` to allocate an object by identity, and `allocateObjectByType` to allocate a random object of the given type. If no objects are available when one of these operations is called, the invocation can wait for a configurable duration. If no objects become available during this duration, the operation will eventually raise an `IceGrid::AllocationTimeoutException` exception. An object allocated by a client cannot be allocated by other clients until the object is released again. Clients can explicitly release objects with the `releaseObject` operation of the `IceGrid::Session` interface. Allocated objects are also released when the client session is destroyed.

Objects can be allocated only if they are declared as allocatable in the server deployment descriptor. To declare an allocatable object, the `allocatable` XML element must be used. This XML element supports the same attributes as the `object` element. For example:

```
// IceGrid XML
<server id="TheServer" exe="./server">
    <adapter name="TheAdapter">
       <allocatable identity="Obj-1"
        type="::App::Encoder">
    </adapter>
  </server>
```

In addition to objects, it is also possible to define *servers* as allocatable. Allocatable servers are useful for security purposes and for the `session` activation mode (see below).

Property sets

The preferred way to configure deployed Ice servers is to use Ice configuration properties. Properties are specified in server descriptors, from which IceGrid generates a configuration file that contains the properties for each server and passes the configuration file via the `--Ice.Config` option on the command line.

With Ice 3.1, IceGrid improves the specification of these properties in descriptors by providing property sets. You can define a set of properties at the application or node descriptor level and include the property set in a server descriptor by using a reference to the set. You can also specify a property set for a specific server or service instance. This allows you to easily change the configuration of a specific server or service template instance without having to change its template descriptor.

Server Activation

On Unix platforms, it is now possible to set the user account under which a server will be executed by the IceGrid node. To use this feature, the IceGrid node must run as root. The user account to use is specified in the server descriptor with the `user` attribute. It is also possible to configure a user account mapper for each node to map the value of the `user` attribute to a specific user account on a specific node. This is useful if a user has different login names on different machines.

We have also added two additional server activation modes:

- `always`: A server with this activation mode is activated as soon as the node is started. The node ensures that the server is always running. If the server is deactivated, the node will re-start it. This activation mode is useful for servers that are always supposed to be running and that cannot use the on-demand activation mode (such as servers without indirect adapters).
- `session`: A server with this activation mode is started on-demand only if it is allocated by a client. A server is allocated when a client allocates one of the allocatable objects in the server. A server that uses the session activation mode cannot be activated otherwise, for example, by attempts to access a non-allocatable object. The server is deactivated again once it is released by the client. This activation mode is useful to run a server with the access privileges of the invoking client, and

can also be used to restrict access to only the user who owns the session.

Improved Security

To use the allocation facility, clients must authenticate with the IceGrid registry. Clients can authenticate using a user name and password, or via a secure connection (SSL). It is important to keep in mind that the allocation facility only enables clients to coordinates access to objects or servers, but does not restrict invocations on objects and servers. (Any client with a proxy of an allocatable object can invoke on the object, even if that client did not allocate the object.)

To secure allocated objects or servers, or in other words, to prevent clients from invoking on objects that they did not allocate, you can combine IceSSL and the session activation mode to restrict access to a server to only the client that allocated the server. You can also use a Glacier2 router to route and filter requests to the deployed servers: an IceGrid session created via a Glacier2 router can only invoke on objects that were allocated by that same session.

Migration

The schema of the IceGrid registry database in Ice 3.1 is incompatible with the IceGrid release in Ice 3.0. To migrate your IceGrid applications, you need to either re-deploy them or upgrade the database. We have provided a simple Python script that uses FreezeScript to upgrade an existing registry database. This script is named `upgradeicegrid.py` and is located in the `config` directory of your Ice distribution. It takes the following arguments:

- the path of the Ice 3.0.x distribution,
- the path of the Ice 3.1 distribution,
- the path of the IceGrid registry database environment to convert; this should be the same path as the path specified by the `IceGrid.Registry.Data` property,
- the path of a directory where the converted database environment will be saved.

For example, you can use the following command to convert the registry database from the IceGrid `simple` demo:

```
/opt/Ice-3.1.0/config/upgradeicegrid.py \
/opt/Ice-3.0.1 /opt/Ice-3.1.0 \
~/Ice-3.0.1-demos/demo/IceGrid/simple/db/registry
~/Ice-3.1.0-demos/demo/IceGrid/simple/db/registry
```

The default templates from the `config/templates.xml` descriptor also changed significantly in Ice 3.1. We have removed most of the optional settings from the template definitions. To set specific configuration properties, you can instead specify them in the server instance property set descriptors.

GUI

The IceGrid GUI in Ice 3.1 saw a number of improvements since version 3.0:

- It is now possible to read application descriptors from files, and save application descriptors to XML files.
- The state of a live deployment (such as the PID of a running server) together with actions on such live deployments (such as starting an idle server) and the editing of application definition are now shown in separate tabs. .
- It is now possible to list, register and unregister dynamic well-known objects directly with the GUI. The GUI also allows you to list and (if desired) unregister dynamically registered object adapters.

The GUI also supports all of the new IceGrid features, such as property-sets, allocatable objects, and the new activation modes.

Changes to IceSSL in Ice 3.1

With the 3.1 release, Ice includes completely revamped C++ and Java implementations of IceSSL, our SSL transport plug-in, as well as a brand-new implementation for C# and Visual Basic in .NET 2.0. Although there are some differences among the platforms, we made every effort to unify their feature sets and configuration options in order to streamline the process of incorporating secure communications into your Ice applications.

General Changes

In previous Ice releases, you were able to configure the plug-in's "client" and "server" behavior independently, where the client configuration affected outgoing connections and the server configuration applied only to incoming connections. This separation made it possible for a single program to use different certificates depending on whether the program was playing the role of client or server in a particular situation, in effect giving the program multiple identities. Upon review, we felt this use case was unlikely to be used in practice and did not justify the extra complexity: In most cases, a program has a single identity regardless of the roles it plays, so we changed IceSSL's configuration scheme for the 3.1 release accordingly. If your program relies on the ability to configure multiple identities, you can still do so with Ice 3.1 by creating a separate communicator instance for each identity.

C++ Migration

The C++ plug-in underwent the most dramatic change because we completely overhauled its configuration scheme. The plug-in no longer uses an XML file to describe its configuration, but rather relies solely on regular Ice configuration properties. Although the XML file provided a degree of flexibility, it was overly complex and difficult to read. The new plug-in offers equivalent functionality, with the added benefit of eliminating the dependency on an external file. As an example of how easily you can configure IceSSL,

let's have a look at the ubiquitous `hello` demo located in `demo/Ice/hello`. The IceSSL properties from the client's configuration file are shown below:

```
# C++ configuration
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=../../../certs
IceSSL.CertAuthFile=cacert.pem
IceSSL.CertFile=c_rsa1024_pub.pem
IceSSL.KeyFile=c_rsa1024_priv.pem
```

The first item of interest is the value of the `Ice.Plugin.IceSSL` property, which has changed in Ice 3.1. This property specifies the entry point for the plug-in that enables the Ice run time to load it dynamically from a shared library or DLL. You will need to update your configuration to use the new entry point.

The remaining properties indicate the names of certificate and key files, as well as the name of a default directory in which to find these files. Although IceSSL supports a number of additional properties, their default values are sufficient for this example, so we don't need to define them.

To assist you in migrating an XML file to the new configuration properties, Ice 3.1 includes the Python script `config/convertssl.py` that accepts the XML filename as an argument and emits equivalent configuration properties to standard output. The resulting properties represent a starting point and may require modification. For example, the script cannot suggest a value for the `IceSSL.DefaultDir` property because its equivalent in prior releases is not specified in the XML file; you will usually need to add a definition for this property.

Java Migration

The removal of separate client and server configurations, as discussed earlier, was the most significant change to the properties in IceSSL for Java, but this release includes other changes as well. For example, the new property `IceSSL.Truststore` takes the place of the old properties `IceSSL.Client.Certs` and `IceSSL.Server.Certs` and specifies the name of a keystore containing trusted certificates. Additionally, in prior releases, the property `IceSSL.Server.ClientAuth` specified how strictly the plug-in should verify a client's credentials; the equivalent property in release 3.1 is `IceSSL.VerifyPeer`.

Another notable addition to this release is a second implementation of the IceSSL plug-in that requires Java 5. The primary advantage of this version of the plug-in is support for the thread pool concurrency model, which scales better than the thread-per-connection concurrency model required by the plug-in for Java 2. The plug-ins are identical in all other respects.

As an example of the new property names, the relevant section from the `hello` client program's configuration file is shown below:

```
# Java configuration
Ice.Plugin.IceSSL=IceSSL.PluginFactory
Ice.ThreadPerConnection=1
IceSSL.DefaultDir=../../../certs
IceSSL.Keystore=client.jks
IceSSL.Password=password
IceSSL.Truststore=certs.jks
```

Note that `Ice.ThreadPerConnection` is present because this configuration file is intended to be compatible with the plug-ins for Java 2 and Java 5. The property `IceSSL.DefaultDir` is new in this release and supplies a default directory in which the plug-in should look for the keystore files identified by the properties `IceSSL.Keystore` and `IceSSL.Truststore`.

.NET Introduction

Microsoft added SSL support in version 2.0 of the .NET Framework, which finally made it possible for us to provide an implementation of IceSSL for C# and Visual Basic. The plug-in's configuration resembles that of its counterparts in Java and C++, as you can see from the property definitions below taken from the `hello` demo's client configuration file:

```
# C# configuration
Ice.Plugin.IceSSL=icesslcs.dll:IceSSL.
PluginFactory
IceSSL.DefaultDir=../../../certs
IceSSL.ImportCert.LocalMachine.AuthRoot=cacert.pem
IceSSL.CertFile=c_rsa1024.pfx
IceSSL.Password=password
Ice.ThreadPerConnection=1
```

As in the other language mappings, the IceSSL plug-in is loaded dynamically; the property `Ice.Plugin.IceSSL` defines the entry point, which in this case is comprised of the name of an assembly and the name of a class.

The plug-in shares other similarities with Java and C++, such as its support for the `IceSSL.DefaultDir` property that defines a default directory in which to look for the files mentioned by other properties. For example, the `IceSSL.CertFile` property defines the name of a file that contains a certificate and its corresponding private key. If the plug-in fails to open the file using the filename given by `IceSSL.CertFile`, it combines that filename with the value of `IceSSL.DefaultDir` and tries again.

This plug-in also has the same concurrency model restriction as the Java 2 implementation in that it requires the thread-per-connection model. The reason is the same in both cases: neither platform supports non-blocking SSL sockets, which are necessary for the thread pool model to function properly.

The most interesting property from the sample configuration shown above is `IceSSL.ImportCert`. The purpose of this property is to install a certificate contained in a file into a certificate store, which is a .NET abstraction representing a collection of certificates. In fact, there are several default certificate stores, which you can browse using the Microsoft Management Console's

snap-in for certificates. While the .NET Framework is negotiating a new SSL connection, it verifies the remote peer's certificate by checking whether it is signed by a Certificate Authority (CA) that the local program considers to be a trusted CA, which it indicates by installing the CA's certificate in a particular store.

In most situations, there is no need for this property because the SSL certificates are signed by a well-known CA such as Verisign. (Alternatively, an organization might use a private CA to create its certificates.) Either way, the CA certificate is likely to already reside in the proper store. In the case of the sample programs included with Ice, their configurations refer to SSL certificates signed by an artificial CA created expressly for use in the examples. This artificial CA's certificate will not be installed in a proper store, and we did not want to require you to do so prior to running the example. As a result, we added `IceSSL.ImportCert` to make it more convenient for to try out a demo that uses SSL, but you may find other uses for this property in your own applications.

Configuring Trusted Peers

While establishing a new connection, the underlying SSL engine performs a number of steps to verify the certificate chain of the remote peer. For example, SSL checks that none of the certificates have expired or been modified, that each certificate in the chain has been properly signed, and that the chain's certificate authority is trusted by the local program. These authentication procedures are vitally important, but applications may want to enforce additional restrictions, such as limiting secure connections to a select group of peers. IceSSL allows you to express such restrictions using the new `IceSSL.TrustOnly` class of configuration properties. Consider the following property definition:

```
IceSSL.TrustOnly=O=My Company
```

This configuration causes IceSSL to reject an SSL connection unless the peer's certificate identifies it as belonging to the organization `My Company`. The property value uses the syntax for distinguished names and can be as specific as you like. Other variations of the property are available that apply restrictions only to outgoing connections, only to incoming connections, or only to incoming connections to a particular object adapter.

API Changes

The C++ plug-in in prior releases supported a number of Slice interfaces that allowed an application to interact directly with the plug-in. We wanted to add this capability to the Java and .NET plug-ins as well, but the existing Slice interfaces were tied too closely to the C++ plug-in. Given the platform-specific nature of SSL-related artifacts such as certificates, we elected to remove these interfaces altogether and replace them with the native types described below.

Plugin Interface

In each language mapping, IceSSL defines a native `Plugin` interface that supports several methods for customizing the plug-in, including the installation of a callback for performing additional verification of peer certificates. An application obtains a reference to the plug-in from the communicator, as shown below in C++:

```
// C++
CommunicatorPtr comm = // ...
PluginManagerPtr pm =
    comm->getPluginManager();
PluginPtr plugin = pm->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin =
    IceSSL::PluginPtr::dynamicCast(plugin);
```

We designed the IceSSL properties with the goal of satisfying the configuration requirements of the majority of applications. However, we realize that some applications will have special needs; therefore, the `Plugin` interface in each language mapping also supports a way to circumvent some of the plug-in's normal property-based configuration. (The amount of customization available varies by language mapping.)

In C++, using this feature requires intimate knowledge of the OpenSSL library and should only be considered as a last resort. A Java application with sophisticated requirements can initialize its own instance of `javax.net.ssl.SSLContext` and pass that to the plug-in, which, for example, might be necessary if the application needs to install special provider implementations for keystores or trust managers. Finally, the .NET plug-in allows an application to supply its own certificate collection, enabling you to collect certificate material in an application-specific manner.

Certificate Abstractions

Java and .NET already provide comprehensive abstractions for X.509 certificates and related types, and IceSSL uses those abstractions in its native interfaces rather than attempting to define its own. In C++, however, a certificate abstraction was necessary to shield applications from dealing directly with OpenSSL. The methods on class `IceSSL::Certificate` were inspired by Java's certificate interface and supply enough functionality to satisfy the needs of most applications.

Certificate Verifiers

When your needs exceed the capabilities of the `IceSSL.TrustOnly` properties, you can configure IceSSL with a custom certificate verifier callback. IceSSL invokes your callback only if the SSL engine successfully completes its authentication process and the connection was not rejected by any `IceSSL.TrustOnly` properties, giving you the final decision on whether to allow the connection to proceed. The callback receives a structure of type `ConnectionInfo` that supplies several attributes of the connection:

- a list of certificate objects representing the peer's certificate chain
- a description of the cipher negotiated for the connections
- local and remote socket information
- a flag indicating whether the connection is incoming or outgoing
- the name of the object adapter that received the connection, if the connection is incoming.

In most cases, only the certificate chain is used to make the determination, while all other attributes are used for informational purposes such as logging.

Obtaining Connection Information

One feature that has often been requested is the ability to obtain information about an SSL connection, and release 3.1 includes this capability in all supported language mappings. Given an instance of `Ice::Connection`, which an application can obtain by calling `ice_getConnection` on a proxy or by using the `con` member of `Ice::Current` in a servant, the plug-in returns an instance of the `ConnectionInfo` structure described in the previous section. The sample code below demonstrates how to obtain information about an SSL connection in Java:

```
// C++
Ice.ObjectPrx proxy = // ...
Ice.Connection conn = proxy.ice_getConnection();
try
{
    IceSSL.ConnectionInfo info =
        IceSSL.Util.getConnectionInfo(conn);
}
catch (IceSSL.ConnectionInvalidException ex)
{
    System.out.println("not an SSL connection!");
}
```

Certificate Authority Tools

Many applications need the added security offered by certificate-based authentication but do not necessarily need an elaborate infrastructure for managing certificates. (This is especially true during the development phase.) To simplify the process of incorporating SSL into your applications, Ice 3.1 includes a number of Python scripts, located in the `config/ca` subdirectory of your Ice installation. These scripts wrap low-level OpenSSL commands and allow you to easily establish a private certificate authority, issue certificate requests, and generate certificates.

Glacier2 Improvements in Ice 3.1

Ice 3.1 includes a number of significant improvements to the Glacier2 router in the areas of filtering, authentication, and session management and prevention of denial of service attacks.

New Authentication Method

The `Glacier2::Router` interface now supports an additional operation to create a session: `createSessionFromSecureConnection`. This operation differs from `createSession` in that it doesn't require a user name or password. Instead, the credentials associated with the SSL connection are used to identify the client, and the new interface `Glacier2::SSLPermissionsVerifier` allows you to authorize sessions created using this new operation. The router supplies a `Glacier2::SSLInfo` structure to your verifier that describes the client's SSL connection, including address information and the client's certificate chain. This information is also provided to the session manager through the new `Glacier2::SSLSessionManager` interface.

Improved Filtering

Prior versions of Glacier2 allowed you to filter requests based solely on the category of object identities. The router's filtering mechanism has been enhanced in Ice 3.1:

- Address Filters: An address filter specifies the host and port, or a range of hosts and ports, that router clients are permitted to use. Address filters control the set of back-end hosts that can be contacted by Glacier2 when forwarding requests invoked using direct proxies. Typically, a Glacier2 router should only be allowed to connect to hosts and ports where Ice servers are running.
- Identity Filters: Whereas a category filter considers only the category of object identities, an identity filter matches the entire identity. The router rejects a request unless the target identity is in the set of allowed identities.
- Adapter Filters: An adapter filter is useful for filtering requests on indirect proxies by limiting invocations to a set of object adapter identifiers.

You can configure these filters statically using configuration properties. Furthermore, the identity and adapter filters can be modified dynamically as described below.

Denial of Service

To prevent denial-of-service attacks from rogue clients, Glacier2 supports a new configuration property `Glacier2.Filter.ProxySizeMax` which limits the memory consumed while managing proxies supplied by router clients.

Session Control

The `create` operation of the `Glacier2::SessionManager` interface now accepts an additional parameter: the proxy of an Ice object implementing the `Glacier2::SessionControl` interface:

```
// Slice
module Glacier2
{

interface SessionControl
{
    StringSet* categories();
    StringSet* adapterIds();
    IdentitySet* identities();
    void destroy();
};

};
```

The session implementation can now destroy a client's session by invoking the `destroy` operation on the `SessionControl` object associated with the session. The `SessionControl` object also allows a session manager to modify category, adapter and identity filters dynamically. Note that the `SessionControl` object is only provided if the administrative endpoints of the Glacier2 router are defined. If these endpoints are not configured, a null proxy is passed to the session manager `create` operation.

Migration

As mentioned in the previous section, the signature of the `create` operation from the `Glacier2::SessionManager` now accepts an additional parameter. You'll need to change your code accordingly (you can ignore this additional parameter if you don't need it).

The `Glacier2.AllowCategories` property has been deprecated. You should use the `Glacier2.Filter.Category.Accept` property instead.

Changes to the Java Mapping in Ice 3.1

Ice 3.1 includes several changes to the Java language mapping, including extended syntax for specifying custom type metadata as well as a new mapping that takes advantage of the language features introduced in Java 5.

Custom Type Metadata

The Slice-to-Java compiler has for some time allowed you to influence the code it generates for sequence and dictionary types by annotating their Slice definitions with custom type metadata. For example, the metadata in the following Slice definition overrides the default mapping for the sequence type `StringList`:

```
// Slice
["java:type:java.util.LinkedList"]
sequence<string> StringList;
```

As a result of this annotation, all occurrences of `StringList` are represented in the generated code by `java.util.LinkedList` instead of the default mapping to a native Java array. This ability to tailor the Java mapping to meet the needs of your application is a powerful and convenient feature, and we have enhanced it further in Ice 3.1.

Java's collection classes in the `java.util` package exhibit the polymorphic design inherent to object-oriented frameworks in that they use interfaces to define the essential functionality, and concrete subclasses to provide various implementations. Unfortunately, the custom type metadata feature in previous Ice releases did not allow you to express this relationship because it required that you specify the name of a concrete collection class. Let us continue the `StringList` example by adding an interface:

```
// Slice
interface Search
{
    bool containsKeywords(StringList keywords);
};
```

The use of a concrete class causes the generated proxy method for `containsKeywords` to have the following signature:

```
// Java
boolean containsKeywords(
    java.util.LinkedList keywords)
```

We would actually prefer that the argument to `containsKeywords` use the abstract type `java.util.List`, which would allow us to pass any subclass of that abstract type and not just an instance of `java.util.LinkedList`. We cannot simply change our metadata to use an abstract type instead because the Slice compiler must be able to generate code that instantiates a collection class and therefore requires the name of a concrete class. In other words, there are some situations where the Slice compiler must emit code such as this:

```
// Java
value = new custom-type();
```

However, these occurrences are relatively rare, so Ice 3.1 adds the ability for you to specify an optional abstract type in your metadata using the following syntax:

```
// Slice
java:type:concrete-type[:abstract-type]
```

We can now rewrite our definition of `StringList` to use this new feature:

```
// Slice
["java:type:java.util.LinkedList:java.util.List"]
    sequence<string> StringList;
```

In fact, we don't really need to add the abstract type in this case because the Slice compiler now uses `java.util.List` as the default abstract type for any custom sequence type that does not explicitly define one. In any event, the generated code will use the abstract type for all occurrences of `StringList`, except when an instance of the type must be created.

For Slice dictionary definitions, the compiler uses `java.util.Map` as the default abstract type, and continues to use `java.util.HashMap` as the default concrete type.

Java 5 Mapping

Ice 3.1 for Java adds support for an alternate language mapping that abandons compatibility with Java 2 in order to make use of the new language features in Java 5. When activated, the Slice compiler maps Slice `enum` definitions into Java's `enum` type and, by default, maps dictionary definitions to instances of the generic type `java.util.HashMap<KeyType, ValueType>`. Slice sequence types remain unaffected and continue to map to native Java arrays for performance reasons.

A new global metadata directive enables the Java 5 mapping, as shown in the following example:

```
// Slice
[["java:java5"]]
dictionary<string, int> StringMap;
```

In general, however, we recommend using the new compiler option `--meta` instead:

```
slice2java --meta java:java5 StringMap.ice
```

We prefer the compiler option for two reasons:

1. If you decide to use the Java 5 mapping, you must use it for all of your Slice definitions, therefore using the compiler option is more convenient than modifying every Slice file to add the proper metadata.
2. If you must also continue to support Java 2 applications with the same Slice files, it is better to define the Java 5 metadata externally using the compiler option.

Users of prior Ice releases could use custom type metadata to accomplish some of the Java 5 mapping, as shown in the following example:

```
// Slice
["java:type:java.util.HashMap<String, Integer>"]
    dictionary<string, int> StringMap;
```

This particular definition is redundant in the 3.1 release because the custom type we specified is now the default mapping for a Slice dictionary. More precisely, `java.util.HashMap<K, V>` is the default mapping for a Slice dictionary's concrete type, while `java.util.Map<K, V>` is the default abstract type.

The Java 5 mapping also supports an optional abstract type in custom type metadata, as described in the previous section. However, the Java 5 mapping uses `java.util.List<T>` as the default abstract type when one is not specified in the custom type metadata of a Slice sequence. For example, the following definition uses an instance of a generic type for a sequence:

```
  // Slice
["java:type:java.util.LinkedList<String>"]
       StringList;
```

All occurrences of `StringList` in the generated code are represented by `java.util.List<String>`, except when an instance must be created, in which case the compiler uses `java.util.LinkedList<String>` instead.

If you used custom type metadata in the past, it is worthwhile to review your Slice definitions to see if your metadata is still necessary and compatible with the new semantics of the Java 5 mapping. Furthermore, you may encounter "unchecked" warnings from the Java compiler after upgrading to Ice 3.1 and activating the Java 5 mapping. These warnings are the compiler's way of notifying you about situations where you are attempting to convert between an untyped class such as `java.util.List` and a generic type instance such as `java.util.List<String>`. If you are fully committed to using Java 5, we recommend modifying your applications to eliminate these warnings.

Distribution Notes

The binary distributions of Ice 3.1 include two versions of the Ice for Java run time. The run time in `lib/Ice.jar` uses the default mapping that is compatible with both Java 2 and Java 5. The run time in `lib/java5/Ice.jar` uses the Java 5 language mapping for all of the Slice definitions included with Ice, including those for services such as IceGrid and Glacier2, and therefore requires a Java 5 environment.

API Changes in Ice 3.1

Ice 3.1 changes a number of APIs. We have had these changes in mind for quite some time and allowed them to accumulate instead of making a few changes with each release, to minimize the number of disruptions for developers. Most of the previous APIs are still there, so you don't have to change your code immediately. We will continue to provide the old APIs for at least another full release cycle, so you have two major releases before the old APIs will disappear for good. The changes make the Ice APIs more consistent and less error-prone, so we believe that you will appreciate this change for the better. This article provides a brief overview of what has changed.

Communicator Initialization

The communicator initialization (`Ice::initialize`) used to come in several flavors to allow you to optionally specify properties and a logger for a communicator. For example, in C++, the function prototypes looked as follows:

```
// C++
CommunicatorPtr initialize(int&, char*[]);
CommunicatorPtr initializeWithProperties(
    int&, char*[], const PropertiesPtr&);
CommunicatorPtr initializeWithLogger(
    int&, char*[], const LoggerPtr&);
CommunicatorPtr
initializeWithPropertiesAndLogger(
    int&, char*[], const PropertiesPtr&,
    const LoggerPtr&);
```

This is somewhat awkward, not only because the function names are long, but also because Ice 3.1 adds a number of new features that can only be set when a communicator is initialized. Rather than continuing to add new initialization functions, we accumulated all these initialization features into a structure that you can pass to `initialize`:

```
// C++
namespace Ice
{
    struct InitializationData
    {
        PropertiesPtr properties;
        LoggerPtr logger;
        StatsPtr stats;
        Context defaultContext;
        // C++ Only
        StringConverterPtr stringConverter;
        // C++ Only
        WstringConverterPtr wstringConverter;
        ThreadNotificationPtr threadHook;
    };
}
```

For languages other than C++, the structure is a class instead and lacks the string converter members, which are specific to C++. As you can see, the structure allows you to specify a number of settings: property settings for the communicator, as well as a logger, statistics collector, and a default context. We have also added a new feature, thread hooks, that you can set when you create a communicator and, for C++, we have added string converters that allow you to automate conversion of strings between UTF-8 encoding and the native codeset used by your program.

The `initialize` function now has the following prototypes:

```
// C++
CommunicatorPtr initialize(
    int&, char*[],
    const InitializationData& =
        InitializationData());
CommunicatorPtr initialize(
    const InitializationData& =
        InitializationData());
```

These overloads for `initialize` allow you to pass nothing, or to pass either an argument vector or an initialization structure (or both). This simplifies initialization because all the settings for a communicator are specified in a single argument. Note that if the argument vector sets properties, and you also specify properties with the `properties` member of the `InitializationData` structure, the settings in the argument vector override the settings in the `properties` member. (The version of `initialize` without an argument vector is useful for programs that want to prevent setting of properties on the command line.)

The new Ice API also removes the `setLogger` and `setStats` operations on the communicator. We did this for efficiency reasons: by only allowing these features to be set once, when a communicator is created, we can cache the values in the Ice run time and do not need to continuously check whether they might have been updated at run time.

For languages other than C++, `initialize` is overloaded similarly, so you get equivalent functionality. For consistency, the `Ice::Service` and `Ice::Application` classes now also allow you to optionally pass an `InitializationData` structure.

Related to communicator initialization are changes in the way properties are processed. The `getDefaultProperties` operation has been removed, and `createProperties` now allows you to optionally pass an argument vector and a default property set. This simplifies the property handling code in applications and makes it easier to write programs that want to explicitly control property values.

Please see the Ice manual for more details on initialization and property handling.

Thread Notification Hook

The thread notification hook is a new feature in Ice. It is supported for all languages and allows you to intercept the creation and destruction of threads created by the Ice run time. The feature is useful if you need to use libraries that require you make thread-specific initialization and finalization calls (such as COM's `CoInitializeEx` and `CoUninitialize`). In order to receive notification of thread creation and destruction, you must implement a callback class and pass an instance of that class in the `threadHook` member of the `InitializationData` structure you pass to `initialize`. The callback class looks as follows:

```
// C++
class ThreadNotification : public IceUtil::Shared
{
public:
    virtual void start() = 0;
    virtual void stop() = 0;
};
typedef IceUtil::Handle<ThreadNotification>
    ThreadNotificationPtr;
```

Your implementation of this class must derive from `ThreadNotification` and implement the `start` and `stop` methods, for example:

```
class MyHook : public ThreadNotification
{
public:
    void start()
    {
        cout << "start: id = "
            << ThreadControl().id() << endl;
    }
    void stop()
    {
        cout << "stop: id = "
            << ThreadControl().id() << endl;
    }
};
```

The Ice run time calls the `start` method as soon as it has created a new thread which can call into user code, and the `stop` method just before a thread exits. The methods are called within the context of the just-created or about-to-be-destroyed thread. To register your callback instance, you pass the corresponding `InitializationData` structure to `initialize`, for example:

```
// C++
int
main(int argc, char* argv[])
{
    // ...
    InitializationData id;
    id.threadHook = new MyHook;
    communicator = initialize(argc, argv, id);
    // ...
}
```

For languages other than C++, your callback class must implement an interface that looks just like the C++ version. Please see the Ice manual for details.

String Converters for C++

For languages such as Java and C#, Ice transparently works with characters from non-English alphabets. However, for historical reasons, this is not the case for C++: C++ provides both narrow and wide strings, and the encoding of these strings is platform-specific.

Ice for C++ 3.1 provides a new metadata directive, `["cpp:type:wstring"]`, that allows you to selectively map Slice strings to `std::wstring` (instead of the default `std::string`). Naturally, depending on which mapping you decide to use, strings must be encoded to match what is used by the underlying platform. For example, the encoding of narrow strings depends on the locale setting as well as the context in which a string appears. (Strings that are used for display purposes are usually encoded as specified by the locale setting (which might require ISO Latin-1), whereas strings that are written to files are often encoded in UTF-8.) Similarly, the encoding of wide strings varies with the platform. (For example, on Windows, wide strings are encoded in UTF-16, whereas with AIX, they are encoded in UTF-16 in 32-bit mode, and in UTF-32 in 64-bit mode.)

By default, if you use the mapping to wide strings, the Ice run time will automatically select an encoding that is appropriate for your program's run-time environment. This means that, if you use wide strings, you typically need not do anything to get the correct behavior. If you use the mapping to narrow strings, the Ice run time delivers (and expects) strings in UTF-8 encoding by default. If you use strings for file I/O, this is usually the correct choice. However, for display purposes, this is guaranteed *not* to work unless you only use characters within the ASCII range or your terminal uses UTF-8 encoding.

Ice for C++ 3.1 allows you to specify separate string converters for narrow and wide strings that you can use to change this default behavior. For example, you could use a wide string converter to have all wide strings internally use JIS X0208 encoding, and you could use a narrow string converter to have all narrow strings internally use ISO Latin-1 encoding. You can implement the converters by implementing a callback interface (similar to the thread notification hook) and register it by setting the appropriate members in the `InitializationData` structure you pass to `initialize`. Once registered, the Ice run time passes all strings through these converters for encoding and decoding. Please see the Ice manual for details on how to implement string converters. (Ice 3.1 also includes a demo in `demo/Ice/converter` that illustrates how to use this functionality.)

Proxy Methods

Unfortunately, Ice is not entirely immune to software rot, and we have allowed some naming inconsistencies to creep into the proxy

methods over time. For example, many (but not all) proxy methods used a `get` prefix for the accessor methods. This led to inconsistencies such as `ice_getIdentity` versus `ice_communicator`. Similarly, some methods used a `new` prefix for creation of a proxy, whereas other did not, for example, `ice_newFacet` versus `ice_router`.

We decided to clean this up and use a consistent naming convention for proxy methods. The Ice 3.1 API consistently uses a `get` prefix for accessors, and drops the `new` prefix for creation methods. Here are the affected methods:

Old Name	New Name
`ice_hash`	`ice_getHash`
`ice_communicator`	`ice_getCommunicator`
`ice_connection`	`ice_getConnection`
`ice_newIdentity`	`ice_identity`
`ice_newContext`	`ice_context`
`ice_newFacet`	`ice_facet`
`ice_newAdapterId`	`ice_adapterId`
`ice_newEndpoints`	`ice_endpoints`
`ice_collocationOptimization`	`ice_collocationOptimized`

In addition several new methods have been added to retrieve data that was previously inaccessible, such as `ice_getLocator`. Please see the Ice manual for full details of these new methods.

Changes in Slice Definitions

Ice 3.1 also changes a few Slice definitions. On the `Communicator` interface, we removed `removeObjectFactory`. One reason is that the operation is redundant: you can always achieve the same thing by setting a flag inside the factory to make it change behavior. Another (and more important) reason is that `removeObjectFactory` caused problems with respect to threading guarantees and exposed the run time to a potential deadlock. With Ice 3.1, an object factory, once set, remains in effect for the life time of its communicator.

The `Communicator` interface also provides two new operations, `stringToIdentity` and `identityToString`. Previously, these functions were provided as static functions. However, in order to allow characters from non-English alphabets to be used in object identities, it must be possible to pass the identities through user-supplied string converters, which are attached to the communicator. As a consequence, `stringToIdentity` and `identityToString` also had to move to the `Communicator` interface. This change is relevant only for C++. For languages other than C++, the static and non-static versions of these functions have identical behavior, and you can use whichever you prefer.

A new operation on the `Ice::Communicator` interface, `createObjectAdapterWithRouter` replaces `addRouter` and `removeRouter`, which we removed. The reason for this change is that changing routers dynamically can cause internal problems in the Ice run time and has dubious semantics. With the new API, the same router for an object adapter remains set for the life time of the adapter.

With previous Ice versions, it was impossible to re-create an object adapter. As of Ice 3.1, you can create an object adapter with the same name as a previous adapter once the call to `waitForDectivate` on the previous adapter has completed.

The `Ice::Current` structure now has a new `requestId` member. For twoway requests, the `requestId` provides the ID of the incoming request; for oneway requests, the ID is zero; for collocated requests, the ID is −1.

The `addProxy` operation on the `Ice::Router` interface has been replaced by `addProxies`. The new operation returns any proxies that are discarded by the router. (This change will not affect you unless you have created a router implementation of your own.)

The `getServerState`, `getServerPid`, `enableServer`, `isServerEnabled`, `startServer`, `stopServer`, `patchServer`, `sendSignal`, `writeMessage` operations from the `IceGrid::Admin` interface can now raise `IceGrid::DeploymentException` if the server can't be deployed on the node. This might happen for example if the server is set to be executed under a given user account and the user doesn't have any account on the node.

The `stopServer` operation from the `IceGrid::Admin` interface can now raise `IceGrid::ServerStopException` if the server is already deactivated or can't be deactivated.

The `getAdapterEndpoints` operation from the `IceGrid::Admin` interface was replaced with the `getAdapterInfo` operation.

The `IceGrid::Session` interface was replaced with the `IceGrid::AdminSession` interface. The new `IceGrid::Session` interface is now for use by IceGrid clients to allocate objects. The `IceGrid::AdminSession` interface is used by administrative clients, such as the IceGridGUI, to manage the registry database and monitor the registry and nodes.

Finally, the entry points for the IceSSL and IceStorm service plug-ins have been renamed from `create` to `createIceSSL` and `createIceStorm`, respectively. This change avoids a name clash if both plug-ins are linked into the same application.

Integrating Ice with a GUI: Part IV

Matthew Newhook, Senior Software Engineer

Introduction

The previous articles in this series on using Ice in a graphical application showed how to send oneway and asynchronous invocations without the risk of blocking the calling thread. In this article, I'll demonstrate the proper way to handle incoming invocations by implementing a shared todo list—similar to the todo list contained in the Google desktop sidebar. The sample application uses the Qt GUI toolkit, but the techniques explored in this article apply equally well to other GUI frameworks.

Interface

First let's design the Slice interface. We can model each element in the todo list as a textual description with a corresponding flag to indicate whether the task has been completed:

```
// Slice
struct TodoItem
{
    int id;
    string desc;
    bool done;
};
```

As you can see, I've also added an `id` field to uniquely identify each item in the list. The need for this field will become apparent when you consider the interface that we'll use to manage the todo list:

```
// Slice
sequence<TodoItem> TodoItemSeq;

exception ItemNotExistException
{
};

interface Todo
{
    void add(string desc);
    void remove(int id)
        throws ItemNotExistException;
    void change(TodoItem item)
        throws ItemNotExistException;
    TodoItemSeq list();
};
```

This interface contains the typical CRUD methods: create, retrieve, update, and delete. Without the `id` field, we could not implement `change` correctly, since we might not be able to locate the desired item using only its description. For example, suppose the server receives two `change` invocations concurrently; the second of the two updates would fail if the first one changes the item's description.

At this point we need to address an important question: How do clients keep their shared view of the todo list synchronized with each other? There are two common methods of synchronization, manual and automatic. Using manual synchronization, the user is responsible for pressing a refresh button to retrieve the latest data from the server. Automatic synchronization, on the other hand, does not require user intervention. Automatic synchronization is much more user friendly, but harder to implement. However, that's why we're here: to solve the hard problems and keep our users happy!

We can use one of two strategies to implement automatic synchronization: polling or pushing. Polling requires the client to contact the server at regular intervals to obtain updates and apply them to the client-side view of the data. Pushing means the server sends updates to the client as they occur. Of the two, pushing is generally considered the better choice because it is more efficient than polling, so we'll implement the push technique.

We'll add a new interface that the server calls whenever changes are made to the content of the todo list.

```
// Slice
interface TodoObserver
{
    void update(TodoItem item);
    void add(TodoItem item);
    void remove(int id);
};

interface Todo
{
    // ...
    void attach(TodoObserver* observer);
    void detach(TodoObserver* observer);
};
```

Consider how an application might (incorrectly) use the observer interface:

```
// C++
TodoItemSeq items = _todo->list();
// put items in list dialog
_todo->attach(new TodoObserverI);
// at this point list is updated
```

This application can lose synchronization if the list is updated after the call to `list` but before the call to `attach`. We can avoid this situation by registering the observer before calling `list`:

```
// C++
_todo->attach(new TodoObserverI);
// at this point list is updated
TodoItemSeq items = _todo->list();
// put items in list dialog
```

Unfortunately, this code also has a problem. It is possible for updates to arrive after the call to `attach` but before processing the result of the call to `list`. Therefore the updates are lost, meaning that the todo list is wrong.

The correct solution is to supply the initial list to the observer when first attached, as shown in the revised interface definition below:

```
// Slice
interface TodoObserver
{
    void initialize(TodoItemSeq items);
    void update(TodoItem item);
    void add(TodoItem item);
    void remove(int id);
};
```

When `Todo::attach` is called, the observer is added to the list of observers and `TodoObserver::initialize` is immediately called with the current dataset. The server must guarantee that `initialize` is called before `update`, `add`, or `remove` is called on the observer. After this change, the GUI client has no further use for the `Todo::list` operation.

Threading Basics

Before diving into the implementation, we should review some basics about the threading models with which Ice and Qt (and indeed many GUI toolkits) operate. Before you sit down to code your application, you should carefully study the documentation of your GUI toolkit to be sure you understand its threading requirements and restrictions.

The typical Qt application that uses Ice has three sets of threads that interest us: the main thread, the client-side thread pool, and the server-side thread pool.

The main thread is the thread that executes `main`. `QCoreApplication::exec` must always be called by this thread, which is also known as the GUI thread since it is the only thread that can perform GUI-related operations. Qt, like many other toolkits, requires all invocations on GUI widgets to be done in the GUI thread.

The Ice run time uses the client-side thread pool to manage outgoing connections (including processing requests received over a bi-directional connection) and to execute AMI callbacks. The server-side thread pool is used to manage incoming connections and to dispatch incoming requests.

To better understand the various threading models, we'll take a look at some code examples.

```
// C++
class TodoDlg : public QDialog
{
public:
    TodoDlg(/*...*/);
private slot:
    void remove();
private:
    const TodoPrx _todo;
};
```

```
TodoDlg::TodoDlg (/*...*/)
{
    //…
    QPushButton* button = new QPushButton(
        "remove", this);
    connect(button, SIGNAL(clicked()),
            this, SLOT(remove());
}

void
TodoDlg::remove()
{
    int id = // ... retrieve id
    _todo->remove(id);
}
```

In this example, the application invokes the `remove` operation from the `remove` slot. Since this slot is connected to the `clicked` signal of a push button, it will always be called by Qt in the GUI thread. As we have discussed in previous articles, invoking a remote operation from the GUI thread is a bad idea because the thread is blocked for the entire duration of the call to the server. Here is a better approach:

```
// C++
class AMI_Todo_removeI : public AMI_Todo_remove
{
public:
    virtual void
    ice_response();
    virtual void
    ice_exception(const Exception& e);
};

void
TodoDlg::remove()
{
    _todo->remove_async(new AMI_Todo_removeI, id);
}
```

Now we don't block the GUI thread at all, assuming that we're using the Ice router technique from my previous article. The calls to `ice_response` or `ice_exception` are made by the Ice run time in a thread from the client-side thread pool, therefore you may not make any direct invocations on a GUI widget from these callbacks.

Now let's consider the observer implementation:

```
// C++
class TodoObserverI : public TodoObserver
{
public:
    virtual void
    update(const TodoItem&, const Current&);
    virtual void
    add(const TodoItem&, const Current&);
    virtual void
    remove(Int, const Current&);
};
```

```
TodoDlg::TodoDlg(/*...*/)
{
    ObjectAdapterPtr adapter =
        communicator->createObjectAdapter(
            "observer");
    _todo->observe(
        TodoObserverPrx::uncheckedCast(
        adapter->addWithUUID(new TodoObserverI)));
}
```

The Ice run time delivers invocations on the `TodoObserverI` servant from a thread in the server-side thread pool. Once again, these methods must not make any direct invocations on a GUI widget.

In summary, under no circumstances should you block the GUI thread. This means no twoway invocations! If you do, the application might become unresponsive and users will get frustrated and annoyed. This issue was the subject of extensive scrutiny in previous articles, so it is not explored in further detail. The client presented in this article uses an Ice router to allow transparent, non-blocking remote invocations from the GUI thread.

Inter-thread Communication

To notify a GUI widget that an event has occurred, such as an incoming request or a reply to an outgoing request, we need the cooperation of the GUI toolkit. This section describes a number of techniques for ensuring that only the GUI thread performs the required actions.

Signals/Slots

In Qt, we generally use a signal to notify a GUI widget about an event. The signal is then connected to a slot, which is called by the Qt run time when the signal is emitted. Prior to Qt 4.0, this mechanism could not be used between threads—that is, emitting a signal outside the GUI thread was illegal, and the results of doing so were not defined. As of Qt 4.0, the guys at Trolltech added a mechanism that allows signals and slots to cross thread boundaries: queued connections between signals and slots. In contrast with a direct connection, where emitting a signal causes the connected slots to be called immediately, a queued connection sends the signal to the connected slots from the thread in which the connected object lives.

Queued signals enable us to notify a widget that an event has occurred using the familiar signal/slot mechanism! Let's run through an example of using queued signals to handle errors from calling `Todo::remove` in an asynchronous callback. First, the AMI callback object must inherit from `QObject`, otherwise it cannot emit a signal. Furthermore, due to a restriction in the Qt meta-object compiler (`moc`), `QObject` must appear as the first base class.

```
// C++
class AMI_Todo_removeI :
    public QObject, public AMI_Todo_remove
{
    Q_OBJECT
public:
    virtual void
    ice_response();
    virtual void
    ice_exception(const Exception& e);
    // ...
};
```

Next we need to add a suitable signal to the AMI callback. It would be ideal to pass the error as an argument to the signal, but what form should the argument take? How about `Ice::Exception`? Qt makes a copy of the arguments to a queued signal, therefore it's not possible to use `Ice::Exception` directly—the argument would be sliced, since the actual exception is an instance of a derived class. To avoid slicing, the argument must be passed by reference or by pointer. In general, passing by reference is not possible for queued events. How about passing by pointer?

```
// C++
signals:
    void error(Exception*);
```

This signal is called as shown below:

```
// C++
void
AMI_Todo_removeI::ice_exception(
    const Exception& e)
{
    emit error(e.ice_clone());
}
```

Who would be responsible for freeing the memory? How about the recipient of the signal?

```
// C++
void
TodoDlg::error(Exception* e)
{
    // Do something with e
    delete e;
}
```

What if there is more than one recipient of this signal? Indeed, what if there is no recipient? Clearly, this approach does not work. A better solution is to create a reference-counted holder object for the exception, and emit that instead.

```
// C++
class ExceptionHolder : public Shared
{
public:
    ExceptionHolder(const Exception& e)
    {
        exception = e.ice_clone();
    }
    ~ExceptionHolder()
    {
        delete exception;
    }
```

```
    const Exception* exception;

private:
    ExceptionHolder(const ExceptionHolder&);
};
typedef Handle<ExceptionHolder>
ExceptionHolderPtr;
```

Next change the signal as follows:

```
// C++
signals:
    void error(ExceptionHolderPtr);
```

The AMI callback can now emit the signal as follows:

```
// C++
void
AMI_Todo_removeI::ice_exception(
    const Exception& e)
{
    emit error(new ExceptionHolder(e));
}
```

For the todo dialog, a corresponding slot must be added.

```
class TodoDlg : public QDialog
{
    // ...
private slots:
    void error(ExceptionHolderPtr);
};
```

Before we can send a signal containing the `ExceptionHolderPtr` type, we must register it with the Qt meta-object system. If we forget this step, Qt won't know how to copy the type, and we'll get an error when trying to emit the signal. The registration step is shown below:

```
TodoDlg::TodoDlg(/*...*/)
{
    qRegisterMetaType<ExceptionHolderPtr>(
        "ExceptionHolderPtr");
    // ...
}
```

By connecting the `error` signal to the appropriate slot on the dialog, then we can send an asynchronous invocation of `remove`:

```
void
TodoDlg::remove()
{
    AMI_Todo_removeIPtr cb = new AMI_Todo_removeI;
    connect(
     cb.get(), SIGNAL(error(ExceptionHolderPtr)),
     this, SLOT(error(ExceptionHolderPtr)));
    _todo->remove_async(cb, id);
}
```

Since the use of queued events requires Qt 4.x, I'll briefly discuss some analogous approaches that are used with other GUI toolkits.

All GUI toolkits that I know of utilize some sort of event queue to process UI notifications. Each of these methods uses the same idea, namely, posting an event to the GUI event queue which then notifies the todo dialog that an error has occurred.

Older Versions of Qt

With older versions of Qt, a custom event can be used to notify the dialog that an error has occurred in a thread-safe manner. The first step is to create a custom event type:

```
// C++
class ErrorEvent : public QCustomEvent
{
public:
    ErrorEvent(const Exception& e)
     : QCustomEvent(QEvent::User) :
        ex(e.ice_clone())
    {
    }

    std::auto_ptr<Exception> ex;
};
```

The custom event is sent from the AMI callback:

```
// C++
void
AMI_Todo_removeI::ice_exception(
    const Exception& e)
{
    qApp->postEvent(
        _todoDialog, new ErrorEvent(e));
}
```

Finally, the dialog class needs a `customEvent` handler:

```
// C++
class TodoDialog : public QDialog
{
// …
protected:
    virtual void customEvent(QCustomEvent*);
};

void
HelloDialog::customEvent(QCustomEvent* e)
{
    ErrorEvent* error =
        dynamic_cast<ErrorEvent*>(e);
    if(error)
    {
        //...
    }
}
```

Microsoft Foundation Classes

With MFC, you can use a custom message to notify the dialog about an error. In particular, the AMI callback posts a user event to the dialog window:

```
// C++
void
AMI_Todo_removeI::ice_exception(
    const Exception& e)
{
    ::PostMessage(_todoDialoghWnd, WM_USER,
        (WPARAM)0, (LPARAM)e.ice_clone());
}
```

The dialog class needs a message handler for the `WM_USER` message:

```
// C++
BEGIN_MESSAGE_MAP(CTodoDlg, CDialog)
// …
    ON_MESSAGE(WM_USER, OnError)
END_MESSAGE_MAP()
```

Finally, the `OnError` method is called when an error occurs:

```
// C+
void
CTodoDlg::OnError(WPARAM wParam, LPARAM lParam)
{
    Exception* e = dynamic_cast<Exception*>(
        lParam);
    // ...
}
```

The Ice distribution contains a complete example of using MFC in the directory `demo/Ice/MFC`.

Java Foundation Classes

In JFC, the simplest approach is to use `SwingUtilities.invokeLater`, which executes the provided `Runnable` argument in the proper thread:

```
// Java
void ice_exception(Ice.UserException e)
{
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run()
        {
            _todoDialog.error(e);
        }
    });
}
```

Implementation

The server implementation is straightforward, so the remainder of this article discusses client-side implementation issues. You can find the source code for the server in the archive that accompanies this article. Let's start with the `TodoObserver` implementation. What should the observer do? Each invocation on the servant is made by the Ice run time from a thread in the server-side thread pool. If a bidirectional connection is used, the invocation would be made by a thread in the client-side thread pool. In either case, the invocation does not occur in the GUI thread, therefore a direct invocation must not be made on any GUI widget. For example, the following code is incorrect:

```
// C++
class TodoObserverI : public TodoObserver
{
public:

    virtual void initialize(
        const TodoItemSeq& data,
        const Current&)
    {
        _todo->initialized(data);
    }
    // ...
};

class TodoDlg : ...
{
public:
    void
    initialized(const TodoItemSeq& data)
    {
        // ... add each item to the list widget
    }
    // ...
};
```

Instead we'll use queued signals—the signal emitted by the servant is queued automatically by Qt to any recipient that was not created in the current thread of control. Since the todo dialog is one such recipient, the emitted signal is received in the GUI thread. The implementation is quite simple:

```
// C++
class TodoObserverI :
    public QObject, public TodoObserver
{
    Q_OBJECT

public:

    virtual void
    initialize(
        const TodoItemSeq& data, const Current&)
    {
        emit initialized(data);
    }
    virtual void
    update(const TodoItem& item, const Current&)
    {
        emit updated(item);
    }
    virtual void
    add(const TodoItem& item, const Current&)
    {
        emit added(item);
    }
```

```
    virtual void
    remove(Int id, const ::Current&)
    {
        emit updated(id);
    }

signals:

    void initialized(TodoItemSeq);
    void added(TodoItem);
    void updated(TodoItem);
    void removed(int);
};
typedef IceUtil::Handle<TodoObserverI>
TodoObserverIPtr;
```

In the constructor of the todo dialog widget, each of these signals is connected to a slot:

```
class TodoDlg : public QDialog
{
    Q_OBJECT

public:

    TodoDlg(
        const TodoPrx&,
        const ObjectAdapterPtr&,
        QWidget *parent = 0);

private slots:
    void error(ExceptionHolderPtr);
    void initialized(TodoItemSeq);
    void added(TodoItem);
    void updated(TodoItem);
    void removed(int);
    //...
};

TodoDlg::TodoDlg(
    const TodoPrx& todo,
    const ObjectAdapterPtr& adapter,
    QWidget *parent) :
    QDialog(parent), //...
{
    // ...
    qRegisterMetaType<TodoItem>("TodoItem");
    qRegisterMetaType<TodoItemSeq>("TodoItemSeq");

    TodoObserverIPtr observerI =
        new TodoObserverI;
    connect(observerI.get(),
            SIGNAL(initialized(TodoItemSeq)),
            this, SLOT(initialized(TodoItemSeq)));
    connect(observerI.get(),
        SIGNAL(added(TodoItem)),
        this, SLOT(added(TodoItem)));
    connect(observerI.get(),
        SIGNAL(updated(TodoItem)),
        this, SLOT(updated(TodoItem)));
    connect(observerI.get(),
        SIGNAL(removed(int)),
        this, SLOT(removed(int)));

    TodoObserverPrx observer =
        TodoObserverPrx::uncheckedCast(
            adapter->addWithUUID(observerI));

    AMI_Todo_observeIPtr cb =
        new AMI_Todo_observeI;
    connect(cb.get(),
        SIGNAL(error(ExceptionHolderPtr)),
        this, SLOT(error(ExceptionHolderPtr)));
    _todo->observe_async(cb, observer);
}
```

Once again, the meta-type information for the `TodoItem` and `TodoItemSeq` must be registered with Qt so that the signal/slot queuing system can correctly copy the types.

Note that the observer is attached in the constructor by an asynchronous call. What would happen if an exception were thrown in the constructor after the observer was attached? What happens if the AMI callback emits an error signal, or the observer servant receives a callback from the todo list server? Wouldn't the signals still be connected to slots in the now-destroyed application widget? In short, the answer is no. When the application widget is destroyed, all connected slots are automatically disconnected in a thread-safe manner.

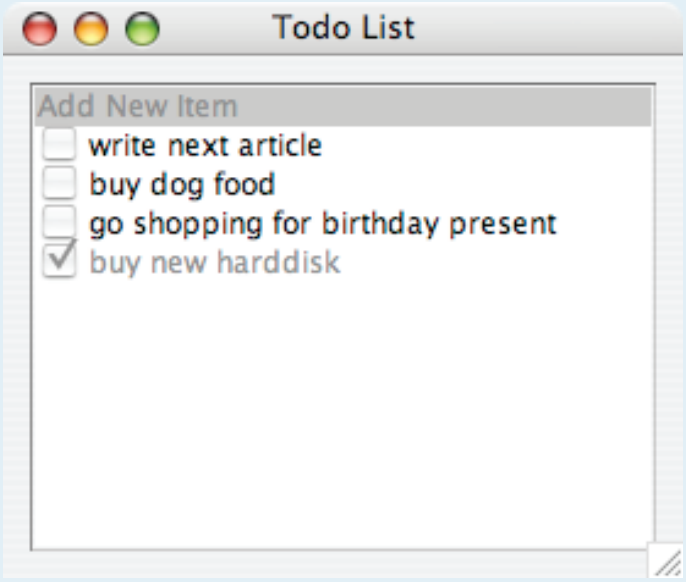
The implementation of `error` is shown below:

```
// C++
void
TodoDlg::error(ExceptionHolderPtr ex)
{
    try
    {
        throw *ex->exception;
    }
    catch(const ItemNotExistException&)
    {
        QMessageBox::information(0,
          "item does not exist",
          "That item has already been removed.");
    }
    catch(const Exception& e)
    {
        ostringstream os;
        os << e;
        QMessageBox::warning(0, "Error",
            os.str().c_str());
        reject();
    }
}
```

This method is called if any remote invocation fails. An occurrence of `ItemNotExistException` results from a conflict with another user of the todo server. For example, if two users concurrently delete a record, one of the users will receive this exception. The client presents `ItemNotExistException` as an informational message box; any other exception is considered fatal and aborts the dialog.

Let's now look at a screen shot of the completed todo application to give you a better idea of how it's supposed to look:

Figure 1: Todo List



Each `TodoItem` is a row in a `QListWidgetItem` in a `QListWidget`. I recommend that you consult the Qt documentation on these classes before continuing. The top row contains a special `QListWidgetItem` that is used to enter new items in the todo list. To delete an item, the user selects the row and presses the Delete key. To edit an item, the user selects a row twice. To edit the top row it is only necessary to select it once. Pressing Escape aborts the edit. Pressing Enter or moving to another row completes the edit.

The classes `AddNewListWidgetItem` and `TodoListWidgetItem` represent the two different types of rows in the `QListWidget`. Furthermore, the `QListWidget` class is subclassed by `TodoListWidget` because some extra functionality is added that can only be achieved by sub-classing.

```
// C++
class TodoDlg : public QDialog
{
    Q_OBJECT

public:

    TodoDlg(
        const TodoPrx&,
        const ObjectAdapterPtr&,
        QWidget *parent = 0);

    virtual void keyPressEvent(QKeyEvent*);

private slots:

    void itemPressed(QListWidgetItem*);
    void currentItemChanged(QListWidgetItem*,
        QListWidgetItem*);
    void editSubmit();
    void editRevert();
    // …

private:

    const TodoPrx _todo;
    QListWidget* _list;
    QListWidgetItem* _prev;
};
```

The class overrides `keyPressEvent` in order to handle the deletion of items. The `itemPressed` slot is called by the Qt run time when a row is selected, which initiates editing of a row. The `editSubmit` and `editRevert` slots are called when editing is completed and aborted, respectively. The member variable `_prev` keeps track of the last selected item to determine whether editing should commence. The remainder of the constructor is presented below:

```
// C++
TodoDlg::TodoDlg(
    const TodoPrx& todo,
    const ObjectAdapterPtr& adapter,
    QWidget *parent) :
    QDialog(parent),
    _todo(todo),
    _prev(0)
{
    _list = new TodoListWidget(this);
    _list->setSelectionMode(
        QAbstractItemView::SingleSelection);
    _list->setEditTriggers(
        QAbstractItemView::DoubleClicked|
        QAbstractItemView::SelectedClicked);

    connect(_list,
        SIGNAL(itemPressed(QListWidgetItem*)),
        this,SLOT(itemPressed(QListWidgetItem*)));
    connect(_list,
        SIGNAL(currentItemChanged(
            QListWidgetItem*, QListWidgetItem*)),
        this,
        SLOT(currentItemChanged(
            QListWidgetItem*, QListWidgetItem*)));
    connect(_list, SIGNAL(editSubmit()),
        this, SLOT(editSubmit()));
    connect(_list, SIGNAL(editRevert()),
        this, SLOT(editRevert()));

    new AddNewListWidgetItem(_list);

    QVBoxLayout *mainLayout = new QVBoxLayout;
    mainLayout->addWidget(_list);
    setLayout(mainLayout);

    setWindowTitle("Todo List");

    setFocusPolicy(Qt::StrongFocus);
```

```
    setFocus();
    // ...
    AMI_Todo_observeIPtr cb =
        new AMI_Todo_observeI;
    connect(cb.get(),
        SIGNAL(error(ExceptionHolderPtr)),
        this, SLOT(error(ExceptionHolderPtr)));
    _todo->observe_async(cb, observer);
}
```

The last statement registers the observer. Eventually the server invokes `initialize` on the observer, and as a result the `initialized` slot is called on the todo dialog:

```
// C++
void
TodoDlg::initialized(TodoItemSeq data)
{
    TodoItemSeq::const_iterator p;
    for(p = data.begin(); p != data.end(); ++p)
    {
        new TodoListWidgetItem(_list, *p);
    }
}
```

The implementation of `added` is very similar to `initialized`:

```
void
TodoDlg::added(TodoItem item)
{
    new TodoListWidgetItem(_list, item);
}
```

Each item in the todo list is represented by a `TodoListWidgetItem`. Before we move on to the implementations of `updated` and `removed`, let's examine the class definition for `TodoItemWidgetItem`:

```
// C++
class TodoListWidgetItem : public QListWidgetItem
{
public:

    TodoListWidgetItem(
        QListWidget*, const TodoItem&);

    const TodoItem& getItem();
    bool syncWithModel();
    void syncWithItem(const TodoItem&);
    // ...

private:
    TodoItem _item;
};
```

The `syncWithModel` method copies the data from the list widget into the contained item data. The method returns `true` if the item data actually changed, and `false` otherwise. The `syncWithItem` method copies new item data into the model.

With that out of the way, we can look at the implementation of `updated` and `removed`:

```
void
TodoDlg::updated(TodoItem item)
{
    for(int i = 1; i < _list->count(); ++i)
    {
        TodoListWidgetItem* lw =
            dynamic_cast<TodoListWidgetItem*>(
                _list->item(i));
        if(lw == 0)
        {
            continue;
        }
        if(item.id == lw->getItem().id)
        {
            lw->syncWithItem(item);
        }
    }
}

void
TodoDlg::removed(int id)
{
    for(int i = 1; i < _list->count(); ++i)
    {
        TodoListWidgetItem* lw =
            dynamic_cast<TodoListWidgetItem*>(
                _list->item(i));
        if(lw == 0)
        {
            continue;
        }
        if(id == lw->getItem().id)
        {
            _list->takeItem(i);
            delete lw;
            if(_prev == lw)
            {
                _prev = 0;
            }
            break;
        }
    }
}
```

The next method of interest is the slot `editSubmit`, which is called after a list row has been edited. An invocation on this slot means one of two things has happened: a new item has been added (the user edited the top row), or an existing item has been updated.

```
// C++
void
TodoDlg::editSubmit()
{
    QListWidgetItem* item = _list->currentItem();
    if(_list->currentRow() == 0)
    {
        AddNewListWidgetItem* addNew =
            dynamic_cast<AddNewListWidgetItem*>(
                item);
        assert(addNew != 0);
        string txt = addNew->text().toStdString();
        if(txt != "Add New Item" && txt != "")
        {
            AMI_Todo_addIPtr cb =
                new AMI_Todo_addI;
            connect(cb.get(),
              SIGNAL(error(ExceptionHolderPtr)),
              this,
              SLOT(error(ExceptionHolderPtr)));
            _todo->add_async(cb, txt);
        }
        addNew->setText("Add New Item");
    }
}
```

If the top row is currently selected, the user is adding a new item. No action is taken if the row text is empty or it matches the default value. Otherwise, we call `add` asynchronously with the new item text. Finally, we reset the text on the row item to the default value.

Note that the method does not actually add the new item to the list; that action occurs as a result of the observer operation `added`. This process can cause the user interface to appear unresponsive if the link to the server is slow. However, applying updates immediately and then attempting to synchronize when the observer event arrives can be quite tricky, so I generally prefer to use the simple approach unless it really makes the user experience uncomfortable.

The remainder of the `editSubmit` method deals with the changing of a row:

```
// C++
void
TodoDlg::editSubmit()
{
    // ...
    else
    {
        TodoListWidgetItem* todoItem =
            dynamic_cast<TodoListWidgetItem*>(
                item);
        assert(todoItem != 0);
        if(todoItem->syncWithModel())
        {
            AMI_Todo_changeIPtr cb =
                new AMI_Todo_changeI;
            connect(cb.get(),
                SIGNAL(error(ExceptionHolderPtr)),
                this,
                SLOT(error(ExceptionHolderPtr)));
            _todo->change_async(
                cb, todoItem->item);
        }
    }
}
```

The method calls `syncWithModel` to synchronize the `TodoItem` with the model data. If the method returns `true`, then `change` is called asynchronously to reflect this change on the server side.

The `keyPressEvent` method deals with the removal of an item:

```
void
TodoDlg::keyPressEvent(QKeyEvent* e)
{
    if(e->key() == Qt::Key_Delete &&
        _item->currentRow() != 0)
    {
        TodoListWidgetItem* lw =
            dynamic_cast<TodoListWidgetItem*>(
                _list->currentItem());
        assert(lw != 0);
        AMI_Todo_removeIPtr cb =
            new AMI_Todo_removeI;
        connect(cb.get(),
          SIGNAL(error(ExceptionHolderPtr)),
          this,
          SLOT(error(ExceptionHolderPtr)));
        _todo->remove_async(cb, lw->getItem().id);
    }
}
```

If the user pressed Delete and the currently selected row isn't the top row, then `remove` is called on the server. Once again this results in a callback on the observer, which removes the row from the list.

The remainder of the implementation is not presented. I encourage you to review the full source accompanying the article and post any questions you may have on the ZeroC forums.

Conclusion

This article concludes my series on using Ice in a graphical application. If you are unclear on any of the topics I have covered in these articles or would like more in-depth treatment of a subject, please let me know!

FAQ Corner

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

Q: Why does my Ice for Java application run out of memory when sending a large string?

The memory utilization of the Java Virtual Machine (JVM) is affected by several factors. As the Ice run time prepares to send a protocol message, it constructs a temporary buffer to hold the encoded form of the input parameters. With respect to memory usage, the encoding process is roughly equivalent to making a copy of the parameters; as your parameters grow larger, so does the buffer that Ice needs to encode them.

String parameters are especially problematic in Java, for two reasons. First, the immutable nature of Java's string type forces the Ice run time to allocate more memory and make more copies than should really be necessary. Second, there is a mismatch between Java's native string representation and the Ice encoding of a string: Java strings are composed of 16-bit Unicode characters, whereas Ice encodes strings using an 8-bit format (UTF-8). This discrepancy means the Ice run time must always perform a conversion, which requires additional memory allocation.

As a result, applications that send very large strings can easily exceed the JVM's default maximum heap size. If increasing the JVM's heap limit is not an option, there are some alternative strategies you should consider.

The technique we usually recommend is breaking a large dataset into chunks rather than sending it all at once, as explained in this FAQ. Although it's typically used in file transfer applications, the chunking technique is equally useful for transmitting a large string.

Another solution is to send the data as a sequence of strings rather than a single string. For example, each element of the sequence could represent a line of the string data. As with the chunking approach, the goal is to reduce the maximum length of the strings that the Ice run time must process. You can even combine the chunking and sequence techniques for a further reduction in memory consumption.

Q: Why do I have to allocate C++ Slice classes and servant classes on the heap?

The Ice C++ mapping extensively uses reference counting to free you from the burden of managing memory: Once you have assigned a newly-allocated class instance to a `Ptr` variable, you no longer have to worry about memory leaks. Even in the presence of exceptions, it is guaranteed that the memory for the instance will be deallocated correctly. (Thanks to the Ice garbage collector, this is true even in the presence of cyclic references among instances.)

Internally, Ice uses the same `Ptr` reference counting mechanism that it provides to developers, and for the same reasons. (Despite the legendary programming skills of ZeroC's staff, in truth, we are no better at manually managing memory than anyone else…) The reference counting is also visible in the Ice APIs. For example, the `addWithUUID` operation on the `ObjectAdapter` has the following signature:

```
// C++
ObjectPrx addWithUUID(const ObjectPtr&);
```

Note that the parameter type is `const ObjectPtr&`, that is, the operation expects a smart pointer to a class instance that is reference counted. This is a Good Thing™ because it allows you to write the following:

```
// C++
adapter->addWithUUID(new MyClassI);
```

This works because `ObjectPtr` has a constructor that accepts an `Object*` so, at the point of call, the compiler constructs a temporary `ObjectPtr` that it passes to `addWithUUID`. The creation of the temporary raises the reference count of the instance to 1. Internally, the object adapter assigns the passed `ObjectPtr` to another `ObjectPtr` in its ASM (Active Servant Map), which raises the reference count to 2; once `addWithUUID` completes, the compiler destroys the temporary `ObjectPtr` it created earlier, which lowers the reference count of the instance down to 1. This turns out to be very useful because the reference counting guarantees that the instance will not be destroyed until precisely the right time, namely, only after its ASM entry has been removed, and only after all operations that are executing inside the instance have completed. (See *"The Samsara of Objects"* in Issue 14 of *Connections* for how to use this feature to implement thread-safe life cycle operations.)

However, this same mechanism can bite you if you allocate class instances on the stack or in static variables. For example:

```
// C++
MyClassI mc;
adapter->addWithUUID(&mc); // Looming disaster!
```

The problem here is that, eventually, once the reference count of the instance drops to zero, the instance will call `delete this`. However, that is most likely going to be the last you see of your

process: Calling `delete` on a stack-allocated instance is likely to incur the wrath of the operating system, which, if you are lucky, will unceremoniously kill the process and nicely commemorate the event with a tomb stone in the form of a core dump. (A stack trace from the core dump will show some error in the internals of the heap.) If you are unlucky, your program may go off and do completely unexpected and strange things because passing a pointer to `delete` that did not come from `new` causes "undefined behavior" (which is C++ standard legalese for "absolutely anything might happen").

The moral of this story is pure and simple: do not allocate Slice class instances or servant class instances on the stack or in static variables, *not ever*. It simply does not work, period.

So, can you protect yourself from this mistake? The answer is yes, you can, by using a little-known feature of C++: If a class has a protected destructor, instances of the class must be allocated on the heap; attempts to allocate an instance on the stack or in a static variable cause a compile-time error. So, if you have a destructor in your class, simply make it protected. If you don't have a destructor already, simply add an empty protected destructor:

```
// C++
class MyClassI : public MyClass
{
public:
    // . . .
protected:
    virtual ~MyClass() {}
};
```

That simple change now makes it impossible to incorrectly allocate a class instance:

```
// C++
// Compile-time error
MyClassI mc;
// We don't get this far.
adapter->addWithUUID(&mc);
```

As an added benefit, making the destructor protected also prevents incorrect deletion of a class instance:

```
// C++
MyClassI* p = new MyClassI;
// ...
delete p; // Compile-time error
```

Again, this is a Good Thing™: With the Ice C++ mapping, you are not meant to delete things by hand—deleting things is the job of `Ptr` variables, which are much better at it than programmers.

Ideally, the Slice compiler would protect you from incorrect allocation but, unfortunately, it cannot do that: The rules of C++ require the protected destructor to be present in the most-derived instance. (Adding a protected destructor to the `MyClass` base does not help because what is instantiated is the derived class `MyClassI`.) However, `MyClassI` is written by you, not by the compiler, so you have to add the protected destructor yourself. (With the next major release of Ice, the compiler will add a protected destructor to non-abstract Slice classes. However, for abstract Slice classes (that is, Slice classes with operations) and for servant classes, you still have to do this yourself.)

So, you can either adhere to the firm rule of "only allocate instances on the heap" or, if you believe that an ounce of prevention is better than a pound of cure, you can habitually add a protected destructor to all your Slice classes and servant classes. It only takes a few seconds, and the cost in terms of memory footprint and performance is zero.