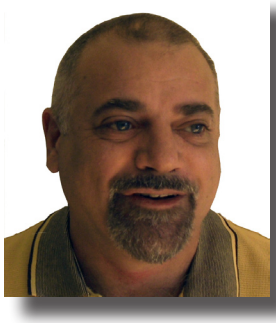
Connections

ZeroC's Newsletter for the Ice Community

Issue Number 28, April/May 2008



Terms of Service

A few months ago and quite by accident, I stumbled across a web page on my ISP's website. Lo and behold, I found that, two months earlier, my ISP had started offering a better Internet plan than my current one for a 20% lower fee. Of course, this was welcome news. What was not so welcome was that the ISP had neglected to tell me (and thousands of other customers) about the new plan and had continued to charge me at the old price for inferior service. I had to call them to get the new plan. (And, no, they did not refund me the difference for the previous two months.)

Two weeks ago (with the same ISP, who shall remain nameless), I ran into an email problem: email to a particular destination was leaving my machine and was accepted by my ISP's SMTP server, but was never delivered. After speaking to the sysadmin of the destination machine, who checked logs and spam filters, it was clear that mail was lost somewhere in between my ISP and the destination machine. Time to call my ISP's support line…

My call ended up at a call center with a voice recognition system. Having chosen technical support, I was not connected to a person, but dropped into a troubleshooting tree. I patiently answered several questions such as "can you browse the web?" to eventually find out that my ISP does not support Thunderbird. (The system kindly offered to help me install Outlook though…) Knowing that the problem was not caused by Thunderbird, I decided to pretend that I *was* using Outlook. However, the troubleshooter has no way to navigate back up the hierarchy (there is no voice equivalent of a "Back" button), so I was stuck at an inappropriate place in the menu system and had to hang up.

On my second call, the system recognized me from my caller ID and asked me whether I still had email problems. I answered "yes", and was helpfully dropped back into precisely the same spot in the troubleshooter that I had just left.

On my third call I was savvy, disabled caller ID, and pretended that I was using Outlook. Soon after, the system asked me to answer a question with "yes" or "no." When I answered "yes", the system did not understand me and asked me to try again. I again answered "yes" but was not understood. After a lecture about background noise (there wasn't any), I got to try a third time. No go—the word "yes" was suddenly no longer in the system's vocabulary. Being told for the fourth time to answer "yes" or "no", I tried saying "no", but the system could not understand that either. At that point, I hung up in disgust.

Not one to give up easily, I decided to use email instead. However, my ISP does not publish an email address. Instead, customers must use a web interface to submit bug reports. I proceeded to do just that, using Safari. After I had filled in endless details, a problem description, and an SMTP trace log, on the final submit, Safari refused to execute a PHP script that was trying to run in the browser and trashed the submission form content. Having tried a second time with the same result, I changed to Internet Explorer. This time, when I pressed the "Submit" button, Internet Explorer locked up.

The following morning, I picked up the phone and cancelled all my contracts with the company.

You may have noticed that we recently released Ice 3.3 beta (with the final release to follow in May). We told you about the release here and in our forum. The software has lots of cool new features and costs exactly the same as always (*nothing* for GPL'd applications), so I encourage you to give it a try. Oh yes—you may have a question or bug report about Ice 3.3 beta. If so, we (the same engineers who wrote the code) will answer you *in person*. We hope you will enjoy both Ice 3.3 beta and being treated like a human being!

Issue Features

New Features and Changes in Ice 3.3

Michi Henning provides an overview of what's new in Ice 3.3.

Background I/O

Benoit Foucher and Mark Spruiell describe the new background I/O mechanism in Ice 3.3.

Highly Available IceStorm

Matthew Newhook explains the new HA features of IceStorm.

Contents

New Features and Changes in Ice 3.3

Michi Henning, Chief Scientist

Introduction

The next release of Ice (3.3) is currently in beta testing. (You can download Ice 3.3 beta from our download page). The final release will follow in May.

For this release, we have added a number of new features and improvements to Ice and its services. In addition, Ice 3.3 removes a number of APIs that have been deprecated for some time, and makes some changes to other APIs and language mappings. This article gives you an overview of what is new, what has changed, and what to watch out for. (Of course, this article only provides an overview—please consult the Ice Manual for full details.)

We invite you to provide your feedback about this release in our discussion forums. Your input and feedback are important to us, and they play a big part in shaping the next release!

New Features

Non-Blocking AMI

In releases prior to Ice 3.2, asynchronous requests could potentially block the calling thread. Even though, in practice, this was unlikely to happen, there was no guarantee that an asynchronous method invocation would never block the caller. Specifically, it was possible for client-side TCP transport buffers to fill up, thereby suspending the caller inside a `write` system call (potentially indefinitely). Apart from this issue, there were other ways for an AMI request to block the caller: during connection establishment, the Ice run time performed a synchronous DNS lookup to resolve the server's domain name, connection establishment itself could block, and, for invocations via indirect proxies, the caller was blocked until the client-side run time had obtained the server's endpoint from the IceGrid registry.

The client could protect itself to some extent from blocking by using invocation timeouts, but doing this is both cumbersome and, of course, presents the problem of choosing an appropriate timeout value. It was also possible to avoid blocking the calling thread of an asynchronous invocation by using a worker thread that performed the invocation in the background. However, doing so required you to write quite a bit of additional code.

We thought it necessary to provide asynchronous invocations that cannot block because, frequently, applications use asynchronous invocations in order to prevent GUI applications from accidentally blocking the GUI thread, thereby freezing the user interface.

As of this release, AMI requests can no longer block the calling thread. The Ice run time initially attempts to synchronously send AMI requests using the caller's thread. This avoids needless context switches to a background delivery thread in the vast majority of cases (namely, when there is no congestion or delays in DNS or IceGrid registry lookups). However, if any of these run-time activities would block the caller's thread, the Ice run time instead queues the request. Delivery of the request is then taken care of by a background thread (for C++ and Java), or using .NET's asynchronous I/O (Ice for .NET).

Of course, you can still use timeouts for asynchronous invocations. Due to the new delivery mechanisms, such timeouts are now more accurate because they no longer depend on the granularity set by `Ice.MonitorConnections`.

This new non-blocking behavior of asynchronous requests, while much more useful than the old behavior, is not completely transparent to your application code, however. For example, you might have an application that sends large numbers of AMI requests to a server. If, for some reason, the server is up, but unresponsive, or there are temporary network problems, the client-side run time will happily queue up as many requests as the application sends. Because each request consumes memory, your application can run out of memory (at least in pathological cases).

To allow your application to deal with this, AMI invocations return a Boolean result that indicates whether the request was queued or sent immediately. In addition, you can arrange for AMI callbacks to be notified when a queued request is sent. This allows you to keep track of the number of queued requests and to limit them to something that your application can tolerate.

Another change in behavior is that an AMI request may now raise `CommunicatorDestroyedException` if another thread concurrently shuts down the Ice run time. Note that this exception is *not* passed to the `ice_exception` callback method, but is raised synchronously at the point you make the call. (Prior to Ice 3.3, it was impossible for an AMI call to raise an exception because all exceptions were reported via the `ice_exception` callback.)

While we were at it, we made a few other improvements to AMI:

- It is now possible to send oneway requests using AMI. For an AMI oneway request, you provide a callback object as usual. However, the `ice_response` method of the callback object is never called (because a oneway request does not return a result and nothing can be known on the client side about when a oneway completes in the server). However, if something goes wrong on the client-side in the process of sending the request, your callback object will be informed of the error via its `ice_exception` method.
- Two new proxy methods, `ice_flushBatchRequests` and `ice_flushBatchRequests_async`, allow you to selectively flush batch requests for individual proxies. The former meth-

od blocks until all queued requests have been handed to the client-side transport, whereas the latter flushes batch requests in the background, without blocking the caller.

- All callbacks for AMI requests are now made by a thread that is taken from an internal thread pool, which guarantees that the caller's thread will *never* be used to invoke `ice_response` or `ice_exception` on a callback object. In turn, this allows you to hold a non-recursive lock while making an AMI request without the potential for deadlock if a callback method needs to acquire the same lock.

You can find more information in Benoit Foucher's and Mark Spruiell's article in this issue.

Improved Glacier2 Scalability

The new AMI semantics allowed us to improve the scalability of Glacier2. In prior releases, Glacier2 used the thread-per-connection concurrency model to isolate clients from each other, so the activities of one client would not adversely affect other clients. However, the thread-per-connection model does not scale as well as a thread pool. As of this release, Glacier2 uses a thread pool to process requests, and uses the new non-blocking AMI to isolate clients from each other.

Buffered delivery of requests now uses at most two threads overall whereas, previously, it required a separate thread for each client. In addition, buffered and unbuffered mode now provide the same degree of isolation of clients from each other; the main reason to use buffered mode now is to allow Glacier2 to deliver oneway requests in batches and to override pending requests. The net effect of these changes is that Glacier2 now scales better and works more efficiently than it did in previous releases.

Transactional Evictor

Freeze now provides a new evictor that automatically encloses all write operations in a transaction. The evictor guarantees ordering of writes, and also allows you to combine several write operations in a single transaction by using collocated invocations. This makes it easier for you to control transaction boundaries when using an evictor and aids data consistency if the database must be recovered after a crash.

IceStorm

IceStorm now provides a high-availability mode that uses master–slave replication with automatic fail-over. Briefly, the replication works by deploying three or more instances of IceStorm on different servers. Logically, all instances are identical replicas, that is, they know about the same topics and subscriptions. The replicated instances use a voting algorithm to elect a master. There is exactly one master at any one time and the remaining instances are slaves. The master processes all changes to topics and subscriptions and replicates these changes to the slaves.

If a slave crashes or becomes disconnected, the remaining slaves and the master continue to forward events. If the master crashes, the remaining slaves automatically elect a new master. If a network failure partitions the redundant IceStorm instances into two or more disconnected "islands", exactly one island elects a new master. (For this to work, the island containing the new master must include a majority of the replicated IceStorm instances, that is, at least $\lceil (n+1)/2 \rceil$ instances must still be able to communicate with each other in order to elect a new master.) Please see Matthew Newhook's article in this issue for more information on the high-availability features of IceStorm.

IceStorm subscriptions are now persistent by default so, if an IceStorm instance crashes and resumes operation again later, subscribers do not need to re-create their subscription for the flow of events to resume.

A new `retryCount` quality-of-service parameter allows subscribers better control over IceStorm's behavior when delivery attempts fail, and IceStorm can now run in a transient mode that maintains no persistent state and therefore does not require a database. (This mode is available only for non-replicated operation.)

Dynamic Network Interfaces

Ice servers that are configured to listen for incoming requests on all network interfaces now bind to `INADDR_ANY` instead of listening separately on each interface. The advantage of doing this is that, if a new interface (such as a wireless connection) becomes available, the server automatically uses it alongside the other interfaces. However, note that an object adapter does *not* automatically adjust the endpoints it publishes in its proxies. Instead, you can call a new method, `refreshPublishedEndpoints` to instruct an object adapter to change the endpoints it publishes in proxies to correspond with the currently available network interfaces.

Support for IPv6 and Multicast

Ice 3.3 allows you to communicate over IPv6 networks. Multihomed hosts that are connected to both IPv6 and IPv4 networks are supported, so a single Ice process can use both transports simultaneously. You can also selectively disable either transport, to control which one will be used by an Ice process.

Servers can now publish multicast addresses in proxies. When a client invokes a oneway operation via multicast proxy, the request is sent to all servers that listen on the corresponding address. Multicast is useful mainly to allow you to build discovery services: by sending a multicast request, clients can reach a server without knowing the exact address of the server. In turn, this allows you to create "zero configuration" clients that dynamically obtain their configuration information on start-up from one of the servers in the multicast group. The only thing that needs to be configured for a client that way is the multicast address for the initial bootstrap request. The Ice distribution includes a demo that illustrates this idea.

Simplified Application Administration

By setting a configuration property, you can instruct the Ice run time to create an additional object adapter that provides a single administrative object to clients. The object has a separate facet for each administrative function it provides. By default, the admin object has two facets: a process facet that is used by IceGrid to instruct a process to terminate cleanly, and a properties facet that allows remote inspection of the configuration settings of a process.

You can add your own facets for your own administrative purposes to this object, and you can control which facets (if any) are to be enabled for a process.

IceBox uses this new feature by adding a facet for its service manager. This allows you to individually start and stop IceBox services from a remote client without additional programming effort.

Ice for .NET

Note that, what was previously known as Ice for C# is now called Ice for .NET: the language independence of .NET really means that Ice can be used with any .NET-enabled language, so it did not make sense to imply one particular language (C#) in the product name.

`slice2vb` Removal

At the time we first supported Visual Basic with Ice, there were no freely-available compilers for either C# or Visual Basic. For that reason, we created a separate `slice2vb` compiler that generates a Visual Basic language mapping—without this, Visual Basic users would have also have required a C# compiler. However, now that compilers for .NET are available free of charge, we decided to drop `slice2vb`.

Of course, this does not mean that you can no longer use Visual Basic with Ice. (Visual Basic is supported just as much as always.) Instead, it means that you must use `slice2cs` to compile your Slice definitions and then compile the generated C# code with a C# compiler. Your Visual Basic application then links with these compiled stubs and skeletons. The Visual Basic demos that ship with Ice include a number of Visual Studio projects that show how to do this.

DLL Name Changes

We have removed the `cs` suffix from the Ice for .NET DLL assemblies and changed their names to align them with the naming conventions for Java and C++. For example, what used to be `icecs.dll` is now `Ice.dll`. Please check the release notes for a complete list of the changed names.

C# Mapping Improvements

The C# mapping now supports generic types for sequences and dictionaries. For example, you can define a sequence and a dictionary as follows:

```
// Slice
["clr:generic:List"] sequence<int> IntSeq;
dictionary<string, string> StringDict;
```

The metadata directive for the sequence causes it to be mapped to the C# type `List<int>` in the `System.Collections.Generic` namespace. For dictionaries, the default mapping uses `Dictionary<string, string>`. You can enable the previous mapping to `CollectionBase` and `DictionaryBase` with a `["clr:collection"]` metadata directive.

Additional metadata directives are available that allow you to map sequences to the `LinkedList`, `Queue`, and `Stack` containers that are provided by .NET, and to map dictionaries to the `SortedDictionary` type.

Note that, as a result of this change, the `Ice.Context` parameter on proxy operation signatures has changed type to `Dictionary<string string>` so, if you are using contexts, you will need to make the corresponding change in your operation implementations.

The new mapping provides not only better type safety, but also better performance for collections of value types because it avoids the cost of boxing and unboxing.

Removed Features

Thread-Per-Connection Concurrency Model

Ice no longer supports the thread-per-connection concurrency model. The main reason for the existence of this model was asynchronous method invocation. However, the new AMI mechanism we introduced with this release has made the thread-per-connection concurrency model obsolete.

If you depend on the ability to serialize requests by using thread-per-connection, you can use the new property *`<threadpool>`*`.Serialize` to get the same serialization semantics.

Java 2

As of this release, Ice no longer supports Java 2 and requires Java 5 or Java 6. (You can still use Ice 3.2.1 or earlier with Java 2; note, however, that ZeroC does not provide support for older releases unless you have a support contract.)

Deprecated APIs

We have removed a number of APIs that have been deprecated for some time. In most cases, if your code still depends on one of these APIs, changing it to use the new APIs will be trivial. Please see the release notes for a complete list of affected APIs.

In addition, Ice 3.3 deprecates a few APIs and properties. (These APIs and properties will be removed completely two minor releases from now, that is, assuming 3.4 and 3.5 are the next two minor releases, these APIs will disappear in release 3.5.) See the release notes for a complete list of the deprecated APIs and properties.

Other Improvements

As usual for a non-patch release, Ice 3.3 contains quite a large number of fixes and improvements. Here are a few selected highlights. (The release notes and the `CHANGES` file in the distribution provide full detail.)

- Collocation transparency now extends to exceptions: for collocated invocations, the caller no longer receives the original exception thrown by the callee, but the same exception it would have received had it made the same invocation in the non-collocated case.
- During communicator destruction, if you set the property `Ice.Warn.UnusedProperties`, the process prints a list of all properties that were set but whose value was never accessed by the program. This makes it easier to track down configuration properties that contain typos.
- The servant locator `locate` and `finished` operations can now throw user exceptions.
- Marshaling performance for .NET applications has been improved.
- C++ servants can now derive from `IceUtil::Thread`.
- Most C++ proxy factory methods now return a proxy of the same type as the original, so you no longer need to down-cast the return value with a `checkedCast` or `uncheckedCast`.
- Ice for .NET now supports signal handling on Mono.
- Ice for .NET can now be compiled as a fully-managed application by defining the `MANAGED` preprocessor symbol. Defining this symbol disables all non-verifyable code, such as P/Invoke calls and `unsafe` constructs. Note that enabling this symbol incurs a minor loss of functionality (for example, protocol compression is not supported) and it incurs a small penalty in marshalling performance.
- Ice for Java now supports the `ICE_CONFIG` environment variable.
- IceBox can now recursively start and stop other IceBox services.
- IceBox services that share a communicator now get a separate communicator instead of using the same communicator as IceBox itself. This means that IceBox services can have their own set of property settings, independent of those set for IceBox itself.
- Object adapters now allow you to specify what proxy options are embedded in the proxies that that are created by an adapter with the `<adapter>.ProxyOptions` property.
- IceGrid provides an improved round-robin load balancing implementation that better handles unreachable servers.

Background I/O

Benoit Foucher, Senior Software Engineer
Mark Spruiell, Senior Software Engineer

Introduction

The Ice run time underwent significant re-engineering for the 3.3.0 release to address two limitations:

- On the client side, oneway and Asynchronous Method Invocation (AMI) calls had the potential to block the calling thread.
- On the server side, a program needed to use the thread-per-connection concurrency model to improve reliability when dealing with misbehaving clients.

Although most applications were not affected by these limitations, the applications that *were* affected had to implement elaborate and complex schemes to work around them.

While implementing the solution to these issues, we referred to the project internally as "background I/O" for reasons that will become clear later in the article. Before we get to that, we discuss the limitations in more detail and give examples of the applications they affected. We'll also describe how to take advantage of this new facility and how to use it efficiently.

Blocking Oneway and AMI Invocations

Prior to Ice 3.3, oneway, batch oneway, and AMI invocations could block under two circumstances. First, an invocation could block while the Ice run time established a connection to the server. This process could be delayed for a number of reasons, including:

- The destination host was unreachable.
- The server was protected by a firewall that silently drops connection requests.
- The server was too busy to accept the new connection.

Second, even if the connection was already established, the invocation could still block if the local transport buffer was full when Ice attempted to write the protocol message.

These issues had significant impact on graphical user interface (GUI) applications. As an example, consider the IceGrid administrative GUI: when the user clicks on the `Stop` button to deactivate a server, the program sends an AMI invocation to IceGrid from the Swing event dispatch thread. The user interface will not respond again until Ice has successfully sent the request. To avoid potentially locking up the GUI, graphical applications typically make invocations from a separate thread; some may take even more drastic measures, such as the ones presented in Matthew Newhook's series of articles "Integrating Ice with GUIs" in Issues 12 to 15 of *Connections*.

These issues also impacted services that forward Ice invocations to clients, such as Glacier2 and IceStorm. For example, IceStorm forwards every message it receives from a publisher to all of its subscribers. If a subscriber becomes unresponsive, IceStorm invocations to that subscriber can block and disrupt the delivery of events to other subscribers. To mitigate this issue, in version 3.2, we implemented a subscriber pool in which a group of threads was tasked to delivering events to subscribers; if an invocation to a subscriber blocked for longer than a configurable timeout period, another thread was added to the pool so that event delivery could continue.

There are situations in which blocking behavior can be advantageous, such as for an application that continuously sends oneway or AMI invocations to a server. In this case, the application can use the blocking semantics for flow control: the calling thread will block as soon as the server fails to keep up with the flow of incoming invocations.

Server-Side Thread Pool Concurrency Model

The responsibilities of the Ice server thread pool include accepting new incoming connections and dispatching invocations from clients. As its name implies, the thread pool concurrency model performs its activities in a thread taken from a dedicated thread pool. This model consumes fewer resources and scales better than the thread-per-connection concurrency model, in which the Ice run time allocates a new thread for each incoming connection.

Prior to Ice 3.3, the server thread pool implementation was not immune to misbehaving clients: a client could cause a thread from the pool to block indefinitely unless a timeout was defined in the server's endpoint configuration.

A thread could also block when using the IceSSL plugin if the client did not respond to the SSL handshake, or it could block if the server sent a large response but the client did not read from its end of the connection: the server's outgoing TCP/IP buffer would fill up and the server thread would block until the client read more data.

If enough such clients were active to exhaust the thread pool, the server was essentially rendered inoperable. This fact could be exploited by malicious clients to craft denial-of-service attacks against Ice servers.

The thread-per-connection concurrency model does not suffer from these vulnerabilities because each connection is managed by a dedicated thread. A malicious client can interfere with the thread associated with its own connection to a server, but cannot disrupt the threads assigned to the server's connections with other clients. However, this concurrency model consumes more system resources (memory and CPU) than a thread pool, which makes it unsuitable for server applications that need to handle a large number of connections.

The Solution: Background I/O

In order to address the issues just described, we carefully analyzed the Ice run time in order to find all operations that might block while processing incoming and outgoing invocations, and we replaced blocking synchronous operations with non-blocking asynchronous operations. For example, we replaced the blocking write that was used to send an AMI request with a non-blocking write; the write operation is completed in the background if necessary to avoid blocking the calling thread.

We also replaced blocking operations such as DNS queries, connection establishment, connection validation, SSL handshaking, Ice locator queries, and Ice router invocations with non-blocking equivalents: if an operation cannot complete without blocking, Ice processes it in the background using a dedicated thread.

With C++ and Java, each communicator creates two threads for handling background I/O tasks: one thread is responsible for monitoring existing connections and executing I/O operations; the other thread performs DNS queries during connection establishment.

With C#, the Ice run time uses the .NET Framework's asynchronous I/O facility for performing socket operations, and a dedicated thread is responsible for executing DNS queries.

Sounds Nice... But How Does This Affect My Ice Client?

To take advantage of background I/O, Ice clients must use AMI or batch oneway requests.

Twoway Invocations

The following invocation using twoway AMI is now guaranteed not to block and can be invoked safely from a GUI event dispatch thread:

```
// C++
class AMICallback : public AMI_Hello_sayHello
{
public:
    void ice_response() {}
    void ice_exception(
                const Ice::LocalException&) {}
};

HelloPrx hello = ...
hello->sayHello_async(new AMICallback());
```

The Ice run time first attempts to send this request from the calling thread. If this cannot be done without blocking, the request is queued and sent in the background instead.

It is important to note that the AMI request is only sent in the background when necessary: small requests that can be sent without blocking are sent by the calling thread, which avoids unnecessary thread context switches and improves performance.

Oneway Invocations

So what about oneway invocations? Consider the following example:

```
// C++
HelloPrx helloOneway = hello->ice_oneway();
helloOneway->sayHello();
```

The semantics of oneway invocations have not changed in Ice 3.3; they are still sent synchronously and may block the calling thread. The reason for this is simple: the application needs to be notified if an error occurs when the Ice run time attempts to send the request. If the oneway invocation were sent asynchronously instead, there would be no way for the application to determine whether the request was sent successfully because the call would return before the run time sends the request. From the application's perspective, the oneway invocation would appear to always succeed, even in the face of serious problems such as an unreachable server.

Does this mean that there is still no way to send a oneway request with the guarantee that it won't block? Fortunately, the answer is no. We have improved AMI to add support for sending oneway invocations asynchronously. It is now possible to write the following:

```
// C++
class AMICallback : public AMI_Hello_sayHello
{
public:
    void ice_response() { assert(false); }
    void ice_exception(
                const Ice::LocalException&) { }
};

HelloPrx helloOneway = hello->ice_oneway();
helloOneway->sayHello_async(new AMICallback());
```

In previous versions of Ice, attempting to invoke the `sayHello` operation asynchronously via a oneway proxy would have resulted in a call to the `ice_exception` callback method with an argument of `Ice::TwowayOnlyException`. This is no longer the case.

In the preceding example, the `ice_response` callback is never invoked by the Ice run time because the server does not send a response for a oneway request. The run time calls `ice_exception` if it encounters an error while sending the oneway request.

Batch Oneway Invocations

A proxy configured for batch invocations allows an application to accumulate a number of oneway requests and send them all at once in a single protocol message. An invocation on a batch proxy looks the same as any other oneway invocation:

```
// C++
HelloPrx helloBatchOneway =
    hello->ice_batchOneway();
helloBatchOneway->sayHello(); // Only queued
```

Internally, however, the Ice run time only marshals and buffers the request; it does not send the batch until the application explicitly flushes it or until the size of the batch reaches the configured maximum message size that prompts the Ice run time to flush it automatically.

Prior to Ice 3.3, queuing an invocation with a batch oneway proxy could block the calling thread while the proxy established a connection to the server. Once the connection was established, further invocations on the proxy were guaranteed not to block. With the addition of background I/O, *all* batch oneway invocations are guaranteed not to block; if a connection is not established yet, the first batch oneway invocation on a proxy causes the Ice run time to create the connection in the background.

You may be wondering what happens to the queued requests if Ice fails to establish the connection in the background. In this scenario, the next invocation on the batch proxy will raise an exception that describes the failure. That exception also serves as an indication that previously-queued requests may not have been sent. Consider the following example:

```
// C++
HelloPrx helloBatchOneway =
    hello->ice_batchOneway();

// If the connection is not yet established,
// it is transparently established in the
// background as a result of the following
// call.
helloBatchOneway->sayHello();

// Obtain the underlying Ice connection
// and close it. ice_getConnection() blocks
// until a connection is established.
helloBatchOneway->ice_getConnection()
    ->close(false);
try
{
    helloBatchOneway->sayHello();
}
catch(const Ice::ConnectionCloseException&)
{
    // Expected, the connection was closed
    // above and some previously-queued requests
    // may have been lost.
}

// The following call should succeed and
// cause the connection to be re-established
// in the background.
helloBatchOneway->sayHello();
```

In this example, the code closes the connection associated with the proxy after it has invoked a batch oneway request, thereby causing the queued request to be lost. The next invocation on the proxy raises an exception to warn the application that a request might have been lost. Any subsequent invocation on the proxy re-establishes the connection in the background.

So, despite the non-blocking guarantee for batch oneway invocations, you must not assume that they will always succeed. If an invocation raises an exception, it indicates that the proxy's connection failed prior to the current invocation.

What about flushing batch requests? Earlier versions of Ice offered two ways to flush batch requests:

- `Communicator::flushBatchRequests`
- `Connection::flushBatchRequests`

Both of these methods block until the batch requests are sent or, more precisely, until they are passed to the underlying transport.

Ice 3.3 adds two new proxy methods for flushing batch requests using synchronous or asynchronous semantics, as shown in the example below:

```
// C++
HelloPrx helloBatchOneway =
    hello->ice_batchOneway();
helloBatchOneway->ice_flushBatchRequests();

class AMICallback :
    public AMI_Object_ice_flushBatchRequests
{
public:
    void ice_exception(
        const Ice::LocalException&) { }
};
helloBatchOneway->ice_flushBatchRequests_async(
    new AMICallback());
```

As you might guess, the `ice_flushBatchRequests` method blocks until the batch has been sent and raises any exceptions directly to the calling thread, whereas `ice_flushBatchRequests_async` never blocks and reports failures via the `ice_exception` callback.

How Does This Affect My Ice Server?

If you are currently using the thread pool concurrency model, there is little you need to do because your server will automatically benefit from the server-side background I/O improvements.

However, there is one point that deserves attention: sending the response of a twoway invocation never blocks the caller. This is important for servers that use AMD to dispatch client invocations, as shown in the following example:

```
// C++
void
HelloI::sayHello_async(
    const AMD_Hello_sayHelloPtr& amdCB,
    const Ice::Current&)
{
    amdCB->ice_response();
}
```

Prior to Ice 3.3, network operations could cause `ice_response` to block, such as when the local TCP/IP buffer was full. This is no longer a concern with the introduction of background I/O: if sending the response would cause the calling thread to block, `ice_response` returns immediately and Ice sends the message in the background instead.

If you were using the thread-per-connection concurrency model to shield your servers from misbehaving clients, you can now change to the thread pool concurrency model. As a matter of fact, we have removed the thread-per-connection concurrency model from Ice 3.3 because background I/O has made it unnecessary.

There was one other use-case for the thread-per-connection concurrency model: ensuring that a connection's oneway requests are dispatched in the order received. This requirement could not be satisfied by the thread pool concurrency model unless you limited the size of the pool to one thread.

We felt this requirement was important enough to justify the addition of a serialization mode to the thread pool. You can enable serialization in a thread pool by setting its `Serialize` property as shown below:

```
Ice.ThreadPool.Server.Serialize=1
```

When enabled, messages from a given connection are processed sequentially in the order they are received, allowing a thread pool with multiple threads to provide the same ordering guarantees as the thread-per-connection concurrency model.

New Client Responsibilities

Flow Control

We mentioned earlier that the blocking behavior of oneway and AMI requests could also be an advantage for flow control: the client's calling thread blocks if the server stops reading messages from its end of the connection.

As we discussed, regular oneway requests still block until the entire message is written to the underlying transport, so nothing has changed in this respect.

For AMI requests, however, the changes made to support background I/O have eliminated this means of flow control. Now, if a server cannot keep up with the flow of incoming AMI invocations from a client, queued requests will pile up indefinitely in the client. In a worst-case scenario, the client will crash after exhausting all available memory.

When do you need to worry about this? This is an issue only for applications that continuously invoke on a proxy. In such cases, it is preferable to use synchronous (non-AMI) invocations to prevent the client from sending too much data and ensure that the server can keep up with the incoming invocations.

However, if you still want to use AMI, you need to implement your own flow control using two new features in Ice 3.3. The first is a change to the signature of all asynchronous proxy methods: in prior releases these methods returned nothing, but now they return a Boolean that indicates whether the request was queued. For example, here is the new signature of the `sayHello_async` method:

```
// C++
bool sayHello_async(
    const AMI_Hello_sayHelloPtr& cb);
```

The method returns true if the request was sent synchronously, that is, the entire message was accepted by the local transport buffer without blocking. If the method returns false, it means the Ice run time queued the request to avoid blocking.

The second new feature is the supplemental AMI callback interface `Ice::AMISentCallback`, which adds the callback method `ice_sent`. The Ice run time invokes this callback once a queued AMI invocation has been sent. In order to receive this callback, your AMI callback class must implement `AMISentCallback` as shown in the following example:

```
// C++
class AMICallback : public AMI_Hello_sayHello,
                    public Ice::AMISentCallback
{
public:
    void ice_response() { assert(false); }
    void ice_sent() { }
    void ice_exception(
        const Ice::LocalException&) { }
};

HelloPrx helloOneway = hello->ice_oneway();
if(helloOneway->sayHello_async(new AMICallback()))
{
    // Request was sent
}
else
{
    // Request was queued; ice_sent() will be
    // called once request is sent.
}
```

Taken together, the return value of asynchronous proxy methods and the `ice_sent` callback allow an application to count the number of currently queued invocations. If that number exceeds some threshold, the application can decide to wait before sending more requests.

AMI Exception Handling

In previous Ice versions, invoking an asynchronous proxy method never raised an exception. If an error occurred, the exception was always reported via the `ice_exception` method of the given AMI callback. Furthermore, the threading semantics of `ice_exception` were never defined; the callback could be invoked directly from the thread calling the asynchronous proxy

method, or it could be called from a thread in an Ice thread pool (or a per-connection thread). Unfortunately, invoking `ice_exception` from the calling thread risked a deadlock if the calling thread locked a non-recursive mutex and `ice_exception` needed to acquire the same mutex.

In Ice 3.3, `ice_exception` is guaranteed not to be called from the calling thread. This allows you to safely acquire mutexes and other resources in the implementation of `ice_exception`.

Another question is what happens if one thread destroys the communicator and a second thread, after the communicator is destroyed, invokes an AMI operation. Because destroying the communicator also destroys the thread pool, the Ice run time no longer has a thread from which it could call `ice_exception` to report that the communicator was destroyed. Instead, the asynchronous proxy method raises `Ice::CommunicatorDestroyedException`, which is the *only* exception that an AMI request can raise *directly* to the calling thread.

In general, most applications will not need to catch this exception, provided they do not invoke on a proxy after they destroy the communicator. Applications that might invoke on a proxy after destroying the communicator should catch the exception:

```
// C++
try
{
    hello->sayHello_async(new AMICallback());
}
catch(const Ice::CommunicatorDestroyedException&)
{
    // The communicator was destroyed.
}
```

Conclusion

Although they look like a small improvement for most applications, the changes to AMI are a big win for other applications. Thanks to background I/O, it is now much simpler to make Ice invocations from a graphical user interface, and your Ice servers can now handle thousands of clients while still consuming very little memory.

Highly Available IceStorm

Matthew Newhook, Senior Software Engineer

Introduction

A prominent new feature of Ice 3.3 is Highly-Available (HA) IceStorm. HA IceStorm uses replication of IceStorm servers to ensure that, if an IceStorm server becomes unavailable for any reason, subscribers and publishers do not suffer an interruption of service. HA IceStorm accomplishes this without backward-incompatible interface changes and, in many cases, IceStorm 3.2 publishers and subscribers can be used without any changes. This article introduces improvements to IceStorm since Ice 3.2.

High Availability

Highly-Available IceStorm uses a master–slave architecture to replicate data. This technique is also used in IceGrid, as described by Benoit Foucher in his article "Master-Slave Replication with Ice" in Issue 23 of *Connections*. I encourage you to read Benoit's article if you'd like more information on this subject, as I am only providing a brief description here.

Consider the following interface:

```
// Slice
interface Counter
{
    nonmutating int getCount();
    void increment();
};
```

This interface contains the operation `increment` for changing the object's state, and the operation `getCount` that reads the state. In typical master–slave architectures, state changes may only occur on the master, while slaves are limited to read-only access. Benoit recommended making this separation explicit in the interface design:

```
// Slice
interface Counter
{
    void increment();
};

interface CounterQuery
{
    nonmutating int getCount();
};
```

Slaves implement the `CounterQuery` interface, and the master implements the `Counter` interface.

This is good advice when designing a replicated service from scratch. However, with IceStorm, we did not have this luxury. IceStorm is a widely-deployed service used in many production systems, and adopting such a change would have required all existing IceStorm applications to undergo substantial modifications.

We opted to keep the existing interfaces intact and, therefore, both the master and the slaves implement the IceStorm interfaces. If an operation that changes state is invoked on a slave, that request is transparently forwarded to the master for processing. In this way, the caller does not need to know which replicas are slaves and which replica is the master.

Replicated Data

So what data does IceStorm replicate?

- The set of topics,
- the subscribers to those topics, and
- the associated subscription quality of service (QoS) data.

For each state change, the master sends a notification to each connected replica. IceStorm does not keep track of publishers, therefore this information is not replicated. Furthermore, links between topics are handled internally in the same way as subscriptions.

Selecting a Master

In IceGrid, the designation of master and slave replicas is configured statically. Designating a new master is a manual process that requires direct intervention by an administrator.

If the IceGrid master becomes unavailable, state-changing operations such as updating the deployment descriptors are no longer possible. However, deployed servers can start and stop, and clients can still be serviced. For IceGrid, this is a reasonable restriction, since the deployment information seldom changes.

With HA IceStorm, the situation is different because, if the master could not change dynamically and were to go down, no state change could take place. Although it might be acceptable to disallow the creation and destruction of topics in this case (because the set of topics is fairly static), for most applications, not permitting new subscriptions would be completely unacceptable as the set of subscribers constantly changes.

The algorithm used by HA IceStorm select a new master is the *Invitation Election Algorithm* as described in "Elections in a Distributed Computing System" by Hector Garcia-Molina. The algorithm works by assigning a priority to each replica. At run time, replicas attempt to gather larger and larger groups of replicas together to form groups; the replica with the highest priority in a group is the leader. Replication can begin once a majority of replicas has been gathered into a group.

The set of available replicas is statically configured; it is not possible to add or remove a replica without bringing down the entire set of replicas.

What is not statically configured, however, is which replica is the master. In HA IceStorm any replica can be the master—this decision is made at run time by a process of voting.

Divergence

Why can replication not begin until a majority of replicas are in a single group? Why is it that a majority is important? A majority is important to prevent partitioning and the subsequent divergence of the IceStorm database that would inevitably occur.

Consider two clients C1 and C2 and two replication groups G1 and G2. C1 talks with group G1, and C2 talks with group G2. (This situation may have arisen because of a network problem, for example.) In this case, replicas in G1 and G2 can no longer talk to each other; consequently, each group could elect its own master and then start replication. Consider now if conflicting state changes are performed by C1 and C2 (for example, C1 creates a topic `foo` in G1 and C2 destroys a topic `foo` in G2). This situation is known as divergence and, once divergence has occurred, the two databases can never be reconciled. Requiring a majority of replicas before replication resumes prevents this problem because only the group containing a majority of replicas will commence replication.

However, requiring a majority implies a number of restrictions:

- HA IceStorm requires at least three replicas to function. If only two replicas were permitted, replication would halt if either of replicas were to fail.
- If no majority can be formed because the network is too fragmented, replication stops.

A full system start-up occurs when no replica is currently running. In this situation, any one of the replicas might hold the most recent database state. To ensure that their databases are synchronized correctly, *all* replicas must be active and in the group before replication can commence. (If instead only a majority were required at system start-up, it would be possible for replication to begin without the participation of the replica with the most recent database, which would obviously cause serious problems.)

IceStorm Clients

IceStorm replicas can have one of the four states listed below:

- `NodeStateInactive`: The node is inactive and awaiting an election.
- `NodeStateElection`: The node is electing a leader.
- `NodeStateReorganization`: The replica group is reorganizing.
- `NodeStateNormal`: The replica group is active and replicating.

For debugging purposes, the state of the replicas can be determined using the `icestormadmin` `"replica"` command (see the Ice Manual for details).

From the perspective of IceStorm clients, however, the replication group is either:

- down: All requests to IceStorm fail.
- inactive: All requests to IceStorm block until the node is either down (in which case the request fails), or becomes active.
- active: Requests are processed.

For any IceStorm client, the first step is to locate a topic of interest using the `TopicManager` interface. The proxy for the topic manager always holds the endpoints of all replicas, either directly or indirectly. (Exactly how this is accomplished is defined by the HA IceStorm deployment; refer to the Ice Manual for more information.) As a result, the topic manager always remains available even if a replica is down, as long as a majority is maintained.

Publishers

From a publisher's point of view, replication works as follows:

- A publisher locates its topic of interest using its configured topic manager.
- The publisher obtains a proxy with which it can publish events.

By default, a publishing proxy obtained by calling `Topic::getPublisher` holds the endpoints of all replicas (directly or indirectly), so that publishing can continue if a replica fails. The publisher normally binds to a single replica and continues to use that replica to publish events unless there is a failure, or until active connection management (ACM) closes the connection.

For many applications, an important requirement is that events are delivered in the same order that they are published. (If the publisher sends events 0, 1, and 2, they must be received by the subscribers in the same order.)

Event delivery order can be guaranteed with HA IceStorm by suitably configuring the subscriber and publisher. The publisher must either:

- publish events using a twoway proxy, or
- publish events using a oneway proxy while the IceStorm publish object adapter has a single thread or is configured to serialize requests.

The subscriber must either:

- subscribe with twoway ordered QoS, or
- use a thread pool that contains a single thread, or use a thread pool that is configured to serialize requests.

In addition, the publisher must use the same replica when publishing events. This point is of critical importance: as soon as a publisher changes to a different replica for publishing events, ordering guarantees are lost. In addition, a publisher may receive no notification that such a change has occurred.

The following circumstances could cause the publisher to transparently change replicas:

- ACM has closed the connection.
- Publishing to a replica fails, and the Ice invocation can be retried. If the invocation cannot be retried, the application receives an exception (see the Ice Manual for full details).
- The server closes the connection due to server side ACM, or because the server shuts down.

There are two options if the publisher absolutely requires notification when it changes to another replica:

- Use `Topic::getNonReplicatedPublisher` to retrieve the publisher proxy. This operation always returns a proxy that contains only the endpoints of the IceStorm replica currently in use.
- Configure the publisher proxy not to be replicated (see the deployment section in the Ice Manual for information on how to do this).

In either case, you can recover from a publishing failure by once again calling `Topic::getPublisher` or `Topic::getNonReplicatedPublisher`. The publisher will rebind to another topic replica, and you will get another publisher proxy with which to publish events.

Subscribers

From a subscriber's point of view, the replication works as follows:

- A subscriber locates its topic of interest using its configured topic manager.
- The subscriber calls `Topic::subscribeAndGetPublisher` on the topic.

The subscription information is automatically forwarded to all replicas so that events published on any replica are forwarded to the subscriber. The subscriber will stop receiving events under two circumstances:

- The subscriber is unsubscribed by calling `Topic::unsubscribe`.
- The subscriber is removed as a result of a failure when delivering events. This is controlled by the retry QoS described below.

Configuration

There is very little to do if you are migrating from IceStorm 3.2 to IceStorm 3.3 and do not want to take advantage of the HA features.

For IceStorm 3.2, we added a subscriber pool to ensure that events were delivered to subscribers without blocking publishers. With the addition of background I/O (see Benoit's and Mark's article in this issue), that pool is no longer necessary, so we removed the following properties:

- `IceStorm.SubscriberPool.Size`
- `IceStorm.SubscriberPool.SizeMax`
- `IceStorm.SubscriberPool.SizeWarn`
- `IceStorm.SubscriberPool.Timeout`
- `IceStorm.Trace.SubscriberPool`

The thread-per-connection concurrency model is no longer supported in Ice 3.3, so we also removed the various thread-per-connection properties:

- `IceStorm.Publish.ThreadPerConnection`
- `IceStorm.Publish.ThreadPerConnection.StackSize`
- `IceStorm.TopicManager.Proxy.ThreadPerConnection`
- `IceStorm.TopicManager.ThreadPerConnection`
- `IceStorm.TopicManager.ThreadPerConnection.StackSize`

We also removed oneway and datagram flush tracing, so the property `IceStorm.Trace.Flush` no longer exists.

All IceStorm properties are prefixed with the service name; consequently, if your IceBox service name (the suffix defined by the property that loads IceStorm, such as `IceBox.Service.`*`<suffix>`*) is not "IceStorm" then you will need to change the names of your properties. For example, consider the following IceBox service configuration:

```
IceBox.Service.Foo=IceStormService,33:createIceStorm ...
```

In this case, the IceStorm configuration properties must use the `Foo` prefix, such as

```
Foo.Discard.Interval=10
```

We also removed the `IceStorm.TopicManager.Proxy` property. This property, although undocumented, was used by `icestormadmin` and the IceStorm example programs to contact the IceStorm topic manager. (Your applications can use whatever method is most appropriate to configure the topic manager proxy.)

You will need to make significant changes to your IceStorm deployment in order to take advantage of replication. Your first decision is whether to use IceGrid or deploy HA IceStorm manually. I recommend that you use IceGrid to deploy HA IceStorm: it is substantially simpler, and it offers the ability to add new IceStorm replicas without having to change the configuration of all of your publishers and subscribers to add the endpoints of the new replicas.

However you deploy HA IceStorm, you will also need to decide the following:

- how many replicas you need (as previously mentioned, the minimum number is three), and
- whether you want `Topic::getPublisher` to return a replicated proxy.

In the remainder of this section, I will focus on deploying HA IceStorm using IceGrid. If you are interested in learning more about manually configuring HA IceStorm, please refer to the Ice Manual.

Ice 3.3 beta did not ship with a pre-defined template for HA IceStorm, but one will be included in the final release of Ice 3.3. This template is shown below:

```
<service-template id="IceStorm-HA">

    <parameter name="instance-name"
      default="${application}.IceStorm"/>
    <parameter name="node-id"/>
    <parameter name="topic-manager-endpts"
     default="default"/>
    <parameter name="publish-endpts"
     default="default"/>
    <parameter name="node-endpts"
     default="default"/>
    <parameter name="flush-timeout"
     default="1000"/>
    <parameter name="publish-rg"/>
    <parameter name="topic-manager-rg"/>

    <service name="IceStorm"
      entry="IceStormService,33b:createIceStorm">
    <dbenv name="${service}"/>
    <adapter name="${service}.TopicManager"
      endpoints="${topic-manager-endpts}"
      replica-group="${topic-manager-rg}"/>
    <adapter name="${service}.Publish"
     endpoints="${publish-endpts}"
     replica-group="${publish-rg}"/>
    <adapter name="${service}.Node"
     endpoints="${node-endpts}"/>

    <properties>
       <property name="${service}.InstanceName"
        value="${instance-name}"/>
       <property name="${service}.NodeId"
        value="${node-id}"/>
       <property name="${service}.Flush.Timeout "
        value="${flush-timeout}"/>
    </properties>
    </service>

</service-template>

<server-template id="IceStorm-HA">
    <parameter name="instance-name"
     default="${application}.IceStorm"/>
    <parameter name="node-id"/>
    <parameter name="topic-manager-endpts"
     default="default"/>
    <parameter name="publish-endpts"
     default="default"/>
    <parameter name="node-endpts"
     default="default"/>
    <parameter name="flush-timeout"
     default="1000"/>
    <parameter name="publish-rg"/>
    <parameter name="topic-manager-rg"/>

    <icebox id="${instance-name}-${node-id}"
      exe="icebox"
      activation="on-demand">
      <service-instance template="IceStorm-HA"
        instance-name="${instance-name}"
        node-id="${node-id}"
        topic-manager-endpts="${topic-manager-end
pts}"
        publish-endpts="${publish-endpts}"
        node-endpts="${node-endpts}"
        flush-timeout="${flush-timeout}"
        publish-replica-group="${publish-rg}"
        topic-manager-replica-group="${topic-manag
er-rg}"/>
    </icebox>
</server-template>
```

The template requires that a replica group be defined for the topic manager; another replica group may also be necessary for use in publisher proxies. Let's say that you decide to have three IceStorm replicas all deployed on the node `localhost`. They would be deployed as follows:

```
<application name="DemoIceStorm">
  <replica-group id="PublishReplicaGroup"/>
  <replica-group id="TopicManagerReplicaGroup">
    <object identity="DemoIceStorm.IceStorm/TopicM
anager"
     type="::IceStorm::TopicManager"/>
  </replica-group>

  <node name="localhost">
    <server-instance template="IceStorm-HA"
     node-id="1"
     publish-rg="PublishReplicaGroup"
     topic-manager-rg="TopicManagerReplicaGroup"/>
    <server-instance template="IceStorm-HA"
     node-id="2"
     publish-rg="PublishReplicaGroup"
     topic-manager-rg="TopicManagerReplicaGroup"/>
    <server-instance template="IceStorm-HA"
     node-id="3"
     publish-rg="PublishReplicaGroup"
     topic-manager-rg="TopicManagerReplicaGroup"/>
  </node>
</application>
```

If no publish replica group is defined, the proxy returned by `Topic::getPublisher` will not be a replicated proxy. In this case, the deployment would change to the following:

```
<application name="DemoIceStorm">
  <replica-group id="TopicManagerReplicaGroup">
    <object identity="DemoIceStorm.IceStorm/TopicM
anager"
     type="::IceStorm::TopicManager"/>
  </replica-group>
```

```
  <node name="localhost">
   <server-instance template="IceStorm-HA"
    node-id="1"
    publish-rg=""
    topic-manager-rg="TopicManagerReplicaGroup"/>
   <server-instance template="IceStorm-HA"
    node-id="2"
    publish-rg=""
    topic-manager-rg="TopicManagerReplicaGroup"/>
    <server-instance template="IceStorm-HA"
     node-id="3"
     publish-rg=""
     topic-manager-rg="TopicManagerReplicaGroup"/>
  </node>
</application>
```

Failures

Some of the HA IceStorm failure scenarios introduce corner cases that your applications may need to deal with.

It is possible for a request to result in an `Ice::UnknownException`. This can happen, for example, if a replica loses the majority of nodes (and thus the node progresses to the inactive state) during request processing. In this case, the result of the request is indeterminate (the request may or may not have succeeded) and the IceStorm client cannot draw conclusions about the status of the request. In this case, your client should retry the request and deal with the potential for failure. For example, consider:

```
// C++
TopicPrx topic = ...;
Ice::ObjectPrx sub = ...;
IceStorm::QoS qos;
topic->subscribeAndGetPublisher(qos, sub);
```

The call to `subscribeAndGetPublisher` may fail with an `Ice::UnknownException`. In this case, the subscription may or may not have failed. Therefore, you should write the code as follows:

```
// C++
IceStorm::QoS qos;
while(true)
{
    try
    {
        topic->subscribeAndGetPublisher(qos, sub);
    }
    catch(const Ice::UnknownException&)
    {
        continue;
    }
    catch(const IceStorm::AlreadySubscribed&)
    {
        // Expected.
    }
    break;
}
```

Persistent Subscribers

Publish–subscribe systems such as IceStorm have two kinds of clients (in the sense of "customer", not in the sense of "Ice client"): publishers and subscribers. Publishers publish information on a topic, and subscribers consume that information. There are, in general, two possible models by which connected parties of such systems can interact. The first is known as the *push model*, in which publishers send (or "push") events to a topic, and the topic pushes events to subscribers. The second is known as the *pull model*, in which topics invoke operations on publishers to retrieve (or "pull") events, and subscribers pull events from the topic.

In the push model, publishers are active because they send events to topics by invoking operations on the topics, that is, publishers act in the role of Ice clients. Subscribers, in contrast, are passive because topics deliver events to subscribers by invoking operations on the subscribers, that is, subscribers act in the role of Ice servers. With the pull model, the situation is reversed: publishers are passive, since topics invoke operations on publishers to retrieve new events (so publishers are servers); correspondingly, subscribers are active because they invoke operations on topics to retrieve new events (so subscribers are clients).

The only model that IceStorm supports is the push model. The pull model (which essentially amounts to polling), is resource-intensive and would be much more complex to implement (both for IceStorm itself and for your publishers and subscribers).

However, why does it matter to think in terms of clients and servers with respect to IceStorm? The distinction becomes important when we consider a crash of IceStorm. A publisher, being a client, discovers the crash as soon as it publishes a new event because the push operation raises an exception. On the other hand, subscribers, which act in the server role, simply no longer receive events. In other words, a publisher can easily identify a crashed IceStorm, but a subscriber does not know why the flow of events has stopped—it might be because publishers are currently not sending any events, or because IceStorm has crashed. (To a subscriber, a dead IceStorm is indistinguishable from a very slow IceStorm.)

In previous versions of IceStorm, topics and information about topic links were persistent, but subscriber information was not persistent. If IceStorm crashed or the network connection between IceStorm and a subscriber was interrupted even temporarily, the subscriber would simply stop receiving events and there was no simple way for a subscriber to tell that this had occurred.

Subscribers can work around this problem, but the techniques to do so are by no means straightforward. For example, one option would be for each subscriber to ping its associated IceStorm topic on a regular basis. If a ping fails, the subscriber marks itself as disconnected and starts a timer to re-subscribe. However, this approach has two problems.The first problem is that a ping call from the subscriber to the IceStorm topic will need to create a new connection, but the ability of the subscriber to connect with IceS-

torm does not imply that IceStorm can connect with the subscriber (for example, due to firewall restrictions). From IceStorm's point of view, the subscriber can be unreachable and therefore remain disconnected.

While you might think this scenario unlikely, there is a more serious problem: IceStorm could crash and restart without the subscriber noticing. For example, suppose the subscriber pings an IceStorm topic every ten seconds. Also suppose that the subscriber pings IceStorm at time *t* and, at time *t*+2, IceStorm crashes. Then, at time *t*+4, IceStorm restarts and, at time *t*+10, the subscriber pings IceStorm again with no problems. However, because IceStorm was restarted, the subscription is lost.

Another option would be for the subscriber to ping the per-subscriber publisher object returned by `Topic::subscribeAndGetPublisher`. Unlike the previous approach, this one is more workable because the ping would fail if IceStorm were to remove the subscription for some reason. Another possible option would be to call `Topic::subscribeAndGetPublisher` on a regular basis: if the subscriber is disconnected, it will be re-subscribed by the call.

None of these workarounds are necessary with IceStorm 3.3 because subscriptions are now stored persistently. Restarting IceStorm does not destroy the subscriber record, and event delivery to all existing subscribers automatically resumes once IceStorm is restarted. (I'll discuss how IceStorm deals with subscribers that forget to unsubscribe shortly.)

However, this new persistence feature may require some changes to your application: subscribers that use fixed identities while assuming that the set of subscribers is cleared on an IceStorm restart may get an unexpected `AlreadySubscribed` exception when calling `Topic::subscribeAndGetPublisher`.

Retry QoS

IceStorm 3.3 adds a new quality-of-service parameter named `retryCount`. This QoS allows a subscriber to influence IceStorm's retry behavior when it attempts to deliver an event and the attempt fails.

There are two types of failure:

- Hard failure: `Ice::ObjectNotExistException` and `Ice::NotRegisteredException`. In the event of hard failure, IceStorm immediately removes subscribers (or linked topics).
- Soft failure: This is any other kind of failure, such as failure to reach the subscriber because its host is down or because no process is listening at the subscriber's endpoint.

In the event of a soft failure, IceStorm suspends event delivery for the time period specified by the `IceStorm.Discard.Interval` property (which defaults to one minute). After a soft failure, IceStorm discards all events for the subscriber for the specified time and resumes event delivery once the interval expires.

Each successive soft failure decrements the configured retry count. IceStorm removes a subscriber once the retry count reaches zero. (A retry count of −1 means retry forever.) Linked topics always have a retry count of −1; the default value of the `retryCount` parameter is 0.

When using a retry count of −1, it is imperative that you configure your applications correctly; otherwise, IceStorm may never remove stale subscriptions because it will continue to attempt delivery until it encounters a hard error. However, hard errors are only possible if IceStorm can actually reach the subscriber or an agent of the subscriber (such as IceGrid). (If the subscriber is down, IceStorm only receives a soft failure.) Therefore, if you use a retry count of −1, each subscriber should use IceGrid or run at a fixed endpoint. In addition, if you expect the subscriber to continue to receive events after it restarts, you must use the same identity for the subscriber on each restart. (You can rely on `Topic::subscribeAndGetPublisher` throwing `AlreadySubscribed` if the subscriber is already subscribed.)

Oneway Delivery Mode

There has been a subtle change in the oneway message delivery mode for subscribers that may affect some users of IceStorm. As you may recall, the message delivery semantics are determined by the proxy that the subscriber passes to IceStorm.

The following example uses twoway delivery because proxies are twoway by default:

```
IceStorm::TopicPrx topic = ...;
ObjectAdapterPtr adapter = ...;
ObjectPrx subscriber = adapter->addWithUUID(
    new HelloI);
IceStorm::QoS qos;
topic->subscribeAndGetPublisher(qos, o);
```

However, the delivery mode becomes oneway if the subscriber supplies a oneway proxy:

```
topic->subscribeAndGetPublisher(
    qos, o->ice_oneway());
```

In IceStorm 3.2, all outgoing events were placed into a queue. Workers from the subscriber pool flushed subscribers on a round-robin basis. When IceStorm flushed a oneway subscriber, it sent the message using a oneway proxy if the queue contained only one event; otherwise IceStorm sent all queued events using a batch oneway proxy and then immediately flushed the connection.

Using a batch proxy increases throughput because it results in a smaller protocol message, makes better use of protocol compression, and results in fewer context switches and system calls.

However, what happens if IceStorm is flooded with events? The result is that the subscriber pool worker threads spend more and more time flushing events, causing the latency of individual events to increase while improving overall event throughput.

Since this is the purpose of the oneway batch mode, we elected to change the semantics of the oneway delivery mode in IceStorm 3.3 to better match the intended semantics. In a nutshell:

- Applications that care more about throughput than latency should use batch mode.
- Applications that care more about lower latency than throughput should use oneway or twoway/ordered delivery.

Latency here refers to individual event latency. That is, what goes up by using batch delivery is the variance of the wait time between events for individual subscribers. However, because throughput is higher, overall latency across all subscribers is reduced.

The upshot of these changes is that applications that used oneway delivery mode with IceStorm 3.2 and rapidly sent a large numbers of messages will see decreased throughput but lower latency.

Database

HA IceStorm introduced a number of changes to its database schema, along with a change from Berkeley DB 4.5 to 4.6. To use existing IceStorm 3.2 databases with IceStorm 3.3, you must first update the databases with the `icestormmigrate` tool. The upgrade instructions are provided in the release notes that accompany the distribution, so please review them for migration instructions.

If you want to migrate to a replicated IceStorm, first migrate the database as described in the release notes, and then copy the resulting database into the database directory of a single replica. (The other replicas should have no database at all.) After doing this, all other IceStorm replicas replicate the database when you start IceStorm for the first time.

Conclusion

The addition of high availability features to IceGrid in Ice 3.2 was extremely important to those users that use Ice in environments that require fault tolerance. In this release, we have continued to improve resilience of Ice applications to failures by making IceStorm fault tolerant as well. I hope that you take advantage of this in your deployments: failure should not be an option!

FAQ Corner

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

Q: Why is it important to use the `idempotent` Slice keyword?

An *idempotent* operation is an operation that, if invoked twice in succession with identical parameter values, has the same effect as a single invocation. For example, the statement `x=1` is idempotent whereas the statement `x++` is not.

Ice allows you to optionally mark a Slice operation as idempotent. For example:

```
// Slice
interface Example
{
    idempotent void setVal(int val);
    void incVal();
};
```

The `idempotent` keyword informs the Ice run time that a single invocation of the `setVal` operation has the same effect as two successive invocations of `setVal` with the same parameter value. Obviously, `incVal` (which adds one to the current value) is not idempotent.

So, why would the Ice run time care about this? The answer is that Ice provides *at-most-once* semantics: it guarantees that, if a client makes a single operation invocation, the invocation will either be delivered to the server exactly once or not at all. This may seem self-evident. However, it implies that under no circumstances will a single invocation by a client ever result in two invocations in the server.

The at-most-once guarantee is important: if Ice were not to provide this guarantee, a single invocation of `incVal` by a client could result in two invocations of the operation in the server, with the net effect that the value would be incremented twice instead of once.

As long as everything works well, the Ice run time does not care if an operation is idempotent or not: call dispatch on the client and server side are exactly the same for idempotent and normal operations. However, when things go wrong, the operation mode (normal or idempotent) becomes important. Consider the following scenario:

1. A client invokes an operation on an object in a server.
2. The client-side run time makes a successful `write` system call on the appropriate socket to send the request.
3. The client-side run time calls `read` on the socket to read the server's reply to the request.
4. The `read` system call returns an error indicating that the connection was lost.

At this point, the client-side run time is in a difficult situation because it cannot know exactly *when* the connection was lost:

- It may have been lost immediately after the `write` system call returned, that is, after the request data was copied into the client-side transport buffers in the kernel. In that case, the data was never sent to the server, but only buffered in the client's machine.
- Alternatively, the connection may have been lost some time after that data was physically transmitted to and read by the server. In that case, the operation was actually executed by the server, but the reply from the server to the client was lost.

In other words, in the preceding scenario, the client-side run time has no idea whether or not the operation ended up executing in the server. Either is possible, and there is no way to find out.

In many cases, the Ice run time will automatically retry a failed request before propagating any error back to the application. This also applies to lost connections: the Ice run time will automatically re-establish lost connections and re-send failed requests, but only if it can guarantee that doing so will not violate at-most-once semantics.

In the preceding scenario, the Ice run time cannot safely re-send the failed request because doing so might end up executing the operation in the server a second time.

If you mark an operation as idempotent, the Ice run time relaxes its conservative rules and is more aggressive in trying to recover from transient errors. In particular, for the preceding scenario, if the operation is idempotent, the run time will attempt to re-establish the lost connection and send the request a second time before propagating any error to the application because `idempotent` tells the run time that is safe to do so.

All this means that, if you use `idempotent` where appropriate, you have a better chance for the Ice run time to transparently recover from errors that, otherwise, you would have to deal with yourself. So, it pays to use `idempotent` on operations for which it is appropriate. In general, these are all operations that only read data in the server, but do not modify it, such as `get` operations, and `set` operations that do not operate on previous state. (Such operations are also known as *stateless* or *context-free* operations.)

Note that life-cycle `destroy` operations are *never* idempotent, even though they look like they are. (See the life cycle chapter in the Ice Manual for a detailed explanation. Briefly, the reason is that, if the client loses connectivity to the server at the wrong mo-

ment, the client can draw false conclusions about the existence of an object.)

Also note that, even if an operation's signature suggests that it is idempotent as far as the client is concerned, the operation may not be idempotent in the server. This usually is the case if you have an operation that reads data in the server in order to return it to the client, but also updates state in the server as a side-effect, for example, to update a statistics counter. In that case (at least if you care about accurate statistics), you should not mark the operation as idempotent. (This is the same as the difference between logical const-ness and physical const-ness in C++; you must decide which is appropriate and implement the operation accordingly.)

Q: Does `idempotent` affect the on-the-wire contract?

In a nutshell, the answer is *yes*. You cannot invoke an operation that is idempotent in the server as a normal operation from a client, and vice-versa. Let us look at an example to illustrate the reason for this. Suppose the server uses the following Slice definition:

```
// Slice
interface Server
{
    void doSomething();
};
```

Whatever it is that `doSomething` actually does, the author decided not to mark the operation as idempotent. (Presumably, the author knew what he or she was doing and had good reasons for this.)

Now the client copies the Slice definition and modifies it as follows:

```
// Slice
interface Server
{
    idempotent void doSomething();
};
```

When the client invokes the operation, it receives a `MarshalException`. This is because the Ice run time marshals whether an operation is idempotent or not as part of the request and, in the server, explicitly checks whether the client's view of the operation matches the server's view. If the server receives an idempotent invocation for a normal operation (or vice-versa), it raises a `MarshalException`.

If Ice would not raise an exception in this case, the client could get away with the preceding trick and, unwittingly, cause damage to the state in the server; by checking that the client's and server's view match, Ice improves type safety and prevents such mistakes.

Another reason for marshaling whether an operation is idempotent is to allow routers (such as Glacier2) to correctly preserve at-most-once semantics: without this information, the router would need access to the Slice definition of all operations that are invoked via the router.

Q: How does Freeze deal with `idempotent`?

In a nutshell, it doesn't: Freeze treats idempotent operations exactly the same as normal operations. That is because both read-only and update operations may be idempotent:

```
// Slice
interface Example
{
    idempotent int getVal();
    idempotent void setVal(int val);
};
```

Both `getVal` and `setVal` are idempotent operations, but only `getVal` is read-only; `setVal` is idempotent even though it modifies state in the server.

What Freeze cares about—or, more precisely, what a Freeze *evictor* cares about—is not whether an operation is idempotent or not, but whether an operation modifies state in the server. (Operations that modify state are known as *mutating* operations; operations that do not are known as *non-mutating* operations.) You can inform Freeze whether an operation is mutating or not with a metadata directive:

```
// Slice
interface Example
{
    ["freeze:read"] idempotent int getVal();
    ["freeze:write"]
    idempotent void setVal(int val);
};
```

These metadata directives mark `getVal` as a non-mutating operation, and `setVal` as a mutating operation. A Freeze evictor needs to know whether an operation is non-mutating in order to decide how to deal with a request—depending on the evictor, it may use different strategies to deal with the database. For example with a transactional evictor, mutating operations are performed in separate transactions whereas non-mutating operations can be completed without accessing the database at all if the data is already in the evictor's cache.

If you do not specify any metadata directive, Freeze assumes that the corresponding operation is non-mutating. It follows that you must mark all operations that are mutating with a `freeze:write` metadata directive.