Connections

ZeroC's Newsletter for the Ice Community

Issue Number 21, February 2007

Ice 3.2 Beta

When you read this, we will have released Ice version 3.2 "beta". In contrast to past releases, we have decided to first release a beta version, shortly followed by a final release. This will give you some time to test the new release and to report problems that can only be solved in a binary-incompatible way. In our past release history we only had one such case, but one is too many.

There have been no changes to the Ice protocol for years, and you won't find any major changes in the rest of the Ice core either. By all standards, the Ice core has become a mature technology. This is important, because it gives you peace of mind when you upgrade.

The bulk of the enhancements are in our add-ons (or "Ice services"). This reflects the natural development cycle of quality middleware and computing frameworks: once there is a mature base technology, the focus shifts towards using this technology in components that provide specialized services to mission-critical applications.

One example is the continued development of IceGrid. Ice applications are growing in complexity and size, with many such applications deployed on computing systems that use a very large number of nodes. IceGrid dramatically simplifies the deployment and management of such applications. Among many other enhancements, a major new feature of IceGrid is the ability to deploy redundant application registries. Removing this last remaining single point of failure has been one of the most requested features, and we are happy to finally offer a solution, delivered with the quality you have come to expect from ZeroC.

The best tools are useless without excellent documentation. Apart from making many improvements to the contents of the Ice manual, we went a step further: our manual is now available not only as a single PDF download, but you can browse the complete manual online as well. In addition, we have a completely redesigned Slice reference available for you.

This issue of Connections is fully dedicated to our new release. Read on, and I'm sure you'll find something that will benefit your application!

Marc Laukien
President
ZeroC, Inc.

Issue Features

What's New in Ice 3.2?

Michi Henning explains changes and additions to the Ice core for release 3.2.

IceGrid Replication

IceGrid now offers a replicated registry to eliminate single-point-of-failure scenarios. Benoit Foucher shows you how to take advantage of this new feature.

IceStorm 3.2

Matthew Newhook gives you the run-down on all the improvements to IceStorm, ZeroC's publish–subscribe service.

Contents

What's New in Ice 3.2? 2

IceGrid Replication 7

IceStorm 3.2 12

FAQ Corner 20

What's New in Ice 3.2?

Michi Henning, Chief Scientist

Introduction

The articles by Benoit Foucher and Matthew Newhook in this issue detail the changes and new features in IceGrid and IceStorm, respectively. This article looks at what else has been added and changed in the 3.2 release—you will probably want to take advantage of at least some of these changes in your code. As always, the 3.2 release also fixes a number of bugs and tweaks Ice in minor ways, which I will not specifically mention here—please see the `CHANGES` file in the Ice distribution for full details.

In the final section, I provide details on changes in documentation, installation, build environment, and supported platforms.

Mapping Changes

C++

The C++ mapping now allows you to map Slice structures to C++ classes (instead of C++ structures). Here is an example:

```
// Slice
["cpp:class"] struct Employee
{
    long number;
    string firstName;
    string lastName;
};
```

By default, `slice2cpp` generates a C++ structure for this definition. The `cpp:class` metadata directive instructs the compiler to generate a C++ class instead:

```
// C++
class Employee : public IceUtil::Shared
{
public:
    Employee();
    Employee(::Ice::Long,
             const ::std::string&,
             const ::std::string&);

    ::Ice::Long number;
    ::std::string firstName;
    ::std::string lastName;

    bool operator==(const Employee&) const;
    bool operator!=(const Employee&) const;
    bool operator<(const Employee&) const;
    bool operator<=(const Employee&) const;
    bool operator>(const Employee&) const;
    bool operator>=(const Employee&) const;
};
```

As for the structure mapping, the generated class contains a public data member for each member of the Slice structure. In addition, the class provides a default constructor as well as a "one-shot" constructor that allows you to instantiate and initialize the class in a single statement:

```
// C++
EmployeePtr e =
    new Employee(1234, "John", "Smith");
```

The class also provides comparison operators that behave like the corresponding operators for the structure mapping: comparisons are performed using the fields of the structure, going from major to minor sort criterion in the order of definition of the structure members.

The motivation for this mapping is that, on occasion, classes can actually be more efficient than structures. This may seem surprising, given that classes require heap allocation and are more complex to marshal than structures. However, if you need to pass a structure by value in many places in your code, you incur the cost of copying the structure whenever it is passed to another function. This copying can be rather expensive, especially for structures with many members of complex type (such as string or sequence members).

The class mapping for Slice structures solves the problem neatly: you can pass a C++ const reference to a smart pointer (`EmployeePtr`) for such a class instance, which, conceptually, is the same as passing the instance by value. This is extremely efficient at run time and, because smart pointers provide reference counting, there are no life time issues: the class instance will be deleted once the last smart pointer to the instance goes out of scope.

Another change to the C++ mapping is that all exceptions now derive from `::std::exception`, so you can catch both standard exceptions and Ice exceptions with a single catch handler.

Non-abstract Slice classes now have a protected default constructor, which makes it impossible to accidentally allocate a class instance on the stack or in static storage.

Stream classes now support the zero-copy API.

Finally, `Ice::initialize` is now overloaded to accept a string sequence instead of an `argc`/`argv` pair. (The same is true for `Ice::Application::main` and `Ice::Service::main`.) This is more convenient if you use application-specific configuration properties because it avoids an extra conversion of the string sequence returned by `parseCommandLineOptions` back into an `argc`/`argv` pair.

C# and Visual Basic

We have added "one-shot" constructors for exceptions. Here is an example:

```
// Slice
exception Error
{
    string msg;
    int val;
};
```

Here is an excerpt of the generated C# code for this exception:

```
// C#
public class Error : Ice.UserException
{
       public int val;
       public string msg;

       public Error();
       public Error(System.Exception ex__);
       public Error(string msg, int val);
       public Error(string msg, int val,
                    System.Exception ex__);
       // ...
};
```

Note that the class has four constructors. The first two of these constructors were present in Ice 3.1, but the second two constructors are new. Both are one-shot constructors that accept one parameter for each data member of the exception, so you can initialize and throw an exception in a single statement:

```
// C#
throw new Error("Good thinking", 99);
```

The second version of the one-shot constructor has a trailing parameter that initializes the `InnerException` property of the `System.ApplicationException` base class that all user exceptions derive from. Note that it is no longer possible to initialize the `Message` property of `ApplicationException`. We removed the ability to do this because a trailing string parameter for one-shot constructors would have caused ambiguities for structures that contain string members (possibly as a combination of base and derived members).

If an exception has a base exception, you still get the one-shot constructors. In this case, they accept one parameter for all data members of the base and derived part, in base-to-derived order.

For Visual Basic, we updated the mapping correspondingly, so you will find the new one-shot constructors there as well.

Core Changes

Improved Configuration

Prior to this release, an IceBox service automatically inherited the property settings of IceBox. For example, if you set `Ice.Trace.Network` for IceBox, any service started by IceBox would inherit that setting. As of release 3.2, this is no longer the case—IceBox services only receive the settings specified by their corresponding service property. For example, assume that we have an IceBox `Weather` service and run IceBox with the following configuration:

```
# IceBox Configuration
IceBox.Service.Weather=WeatherI --Ice.Config=Weath
er.cfg
IceBox.ServiceManager.Endpoints=...
Ice.Trace.Network=1
```

With release 3.1, the weather service would run with network tracing whereas, with release 3.2, the weather service only receives the configuration specified in `Weather.cfg`. For backward-compatibility reasons, you can re-establish the old behavior by setting `IceBox.InheritProperties`. If you have existing services running with IceBox 3.1, it is likely that you will need to set this property to retain their current behavior.

The `Ice.Logger.Timestamp` property is deprecated as of this release—time stamps are now always added to log messages.

A new property, `Ice.Default.EndpointSelection`, allows you to specify an endpoint selection policy for a communicator.

You can use `<adapter-name>.ThreadPerConnection` and `Ice.ThreadPerConnection` to configure the thread-per-connection model for an adapter or communicator.

Thread-per-Connection for Proxies

You can require the connection used by a particular proxy to use a thread-per-connection model, by calling a new method, `ice_threadPerConnection` on a proxy. This method returns a new proxy; the replies to invocations on that proxy are processed using the thread-per-connection model. The `ice_isThreadPerConnection` method on proxies allows you to test whether a proxy is configured in this way.

You can also enable this feature via per-proxy configuration properties, as explained in the following section.

Converting Properties to Proxies

The Communicator interface provides a new operation, `propertyToProxy`, that makes it easy to create proxies from a group of property settings. For example, you could set the following properties for your application:

```
# Node Configuration
MyApp.Proxy=id:tcp -h hostA -p 1207
MyApp.Proxy.PreferSecure=1
MyApp.Proxy.EndPointSelection=Ordered
```

With these settings, calling
`propertyToProxy("MyApp.Proxy")`
on a communicator returns a proxy with the specified settings. This makes configuration of clients more flexible.

Nonmutating Operations

We have deprecated the `nonmutating` keyword. As of this release, you should use `idempotent` instead to mark nonmutating operations. For C++, if you want to map read-only operations to C++ `const` member functions, you can do so by adding the `["cpp:const"]` metadata directive to the corresponding operation.

Note that, if a 3.2 client must interoperate with a 3.1 server that uses `nonmutating` for an operation, but you have replaced `nonmutating` with `idempotent` only in the client-side Slice definitions, you will get an unmarshaling error on the server side. To provide backward compatiblity, you can add the `["nonmutating"]` metadata directive to an operation's Slice definition on the client side, which ensures that the correct mode flag is sent on the wire to prevent the unmarshaling error.

For Freeze, we have added the `["freeze:read"]` and `["freeze:write"]`metadata directives, which you can attach to either an interface (in which case the directive applies to all operations), or to a specific operation (in which case the directive overrides the directive attached to the interface, if any). The directives inform Freeze evictors whether an operation modifies its object, so the evictor can adjust its save behavior accordingly. Without any directive, the evictors assume that all operations are write operations. For backward compatibility, you can set the `Freeze.Evictor.UseNonmutating` property which causes Freeze to assume that nonmutating operations do not modify the state of the target object.

Contexts

We have made changes to the way contexts are established and propagated. (The 3.1 APIs still exist, but are deprecated.) Firstly, the core contains a new interface `ImplicitContext`:

```
local interface ImplicitContext
{
    Context getContext();
    void setContext(Context newContext);
    string get(string key);
    string put(string key, string value);
    string remove(string key);
    bool containsKey(string key);
};
```

In addition, the `Communicator` interface has a new operation:

```
// Slice
local interface Communicator
{
    ImplicitContext getImplicitContext();
    // ...
};
```

The `ImplicitContext` interface allows you to get and set a context dictionary that is associated with a communicator, as well as manipulate individual entries in that dictionary. If you establish an implicit context on a communicator, that context is sent by the Ice run time with every operation invocation you make via a proxy that was created by that communicator. This replaces the per-communicator default context of Ice 3.1.

The new mechanism is more powerful because you can control the scope of the implicit context by setting a property, `Ice.ImplicitContext`. You can set this property to one of three values:

- `None`
- `Shared`
- `PerThread`

Setting the property to `None` (or leaving it unset) means that the communicator has no implicit context, so no context is sent with each invocation (unless you explicitly provide a context at the point of call, or attach a context to a specific proxy).

Setting the property to `Shared` creates a single implicit context, whose contents are sent with every operation invocation. In addition, the operations on the `ImplicitContext` interface are interlocked in this case, so you can manipulate the implicit context from different threads without risking race conditions or data corruption.

Setting the property to `PerThread` creates a separate implicit context for each execution thread that makes invocations. (In effect, there are as many contexts as there are invocation threads in the client, with each context held in thread-specific storage.) This allows you to propagate context values that differ for each thread, which is necessary, for example, to implement distributed transactions.

As with Ice 3.1, you can still pass a context explicitly by supplying an additional context parameter to an individual invocation, and you can still attach a context to an individual proxy. However, the semantics of how contexts interact have changed. Here are the complete rules for Ice 3.2:

- Whenever you supply an explicit context at the point of call, only that context is sent with the invocation; the implicit context (if any) and the per-proxy (if any) context are simply ignored in that case.

- If you have set an implicit context on a communicator and invoke on a proxy without a per-proxy context, the implicit context on the communicator is sent. Changes to the implicit context immediately become visible to all invocations made via proxies created by that communicator. (In effect, if a proxy does not have a per-proxy context, and the invocation was not supplied with an explicit context, the *current* implicit context on the communicator provides the default.)
- If you set a per-proxy context on a proxy, that per-proxy context is stored by the proxy and, thereafter, used for all invocations via that proxy (unless overridden by an explicit context at the point of call). Because proxies are immutable, such a per-proxy context, once set on a proxy, cannot be modified.
- If you have an implicit context on a communicator as well as a per-proxy context on an individual proxy, the two contexts are *combined*, so *both* the implicit context and the per-proxy context are sent with invocations via the proxy. If the implicit context and the per-proxy context contain name–value pairs with the same keys but different values, the name–value pairs of the per-proxy context take precedence.

Logging

Ice has always allowed you to attach a logger to a communicator. However, if your code used multiple communicators, you needed to attach a logger to each communicator. With Ice 3.2, we have added a per-process logger that applies to all communicators (unless a specific communicator uses a separate logger). You establish this per-process logger by calling `Ice::setProcessLogger`, and `Ice::getProcessLogger` returns the current per-process logger (and creates a default logger that logs to `stderr` if no per-process logger has been set previously). Once you have called `Ice::setProcessLogger`, all communicators created after that point automatically use the per-process logger (unless you specify a different logger at communicator creation time).

User-Defined Signal Handlers

The `Ice::Application` class makes it easy to correctly initialize the Ice run time, and to correctly finalize it again, even in the presence of signals. Prior to release 3.2, however, there was no way to do application-specific clean-up on receipt of signal. This release fixes this by providing an `interruptCallback` method that you can override to add application-specific clean-up code that is executed when a signal is received.

Improved Configuration Checking

You can set Ice and Ice service properties on the command line or in a configuration file. Misspelling a property name has always been caught by the Ice run time, provided that the property name starts with one of the recognized prefixes, such as `Ice` or `IceBox`. However, properties that are targeted at an object adapter were not checked for spelling errors prior to this release, so a property setting such as `MyAdapter.AdapterID=1` was not caught as erroneous. As of this release, such a configuration error is reported with a warning. (If you are finding it hard to see what is wrong with the preceding setting, you now know why we made this change—`AdapterId` must have a lower-case '`d`'.)

Similarly, prior to release 3.2, it was easy to misspell the name of an object adapter, resulting in no configuration of the adapter. For example, an error such as `MyAdater.AdapterId=1` went undetected. As of this release, if you create a named object adapter without any configuration, the run time raises an `InitializationException`.

Another potential source of error was accidental creation of an object adapter without endpoints due to such a spelling mistake. With release 3.2, the run time prints a trace message for object adapters without endpoints if you set `Ice.Trace.Network` to 2, making it easier to diagnose such a problem.

Testing for Run Time Finalization

The `Communicator` interface now contains an operation `isShutdown`, and the `ObjectAdapter` interface now contains an operation `isDeactivated`. With these operations, you can write code that must be aware of finalization without having to track the state of the run time in separate variables.

Flushing Batched Message

In releases prior to 3.2, it was easy to exceed the maximum allowable message size of a batch before flushing the batch. With Ice 3.2, batch messages are flushed automatically once the addition of another message to the batch would cause the batch to exceed the maximum message size, so you no longer need to worry about flushing explicitly before the maximum is exceeded. You disable this behavior by setting `Ice.BatchAutoFlush` to zero. Explicit flushing is useful if you need to guarantee in-order dispatch of messages across groups of batches—see the Ice Manual for details.

Selection of Secure Endpoints

If you set the property `Ice.Default.PreferSecure`, the run time attempts to contact an object via its SSL endpoints before trying any TCP (or UDP) endpoint. You can also set this behavior on a per-proxy basis by calling `ice_preferSecure` on a proxy, and test whether a proxy is configured that way by calling `ice_isPreferSecure`.

You can also set `Ice.Override.Secure`, in which case the run time will ignore all non-SSL endpoints in a proxy.

Documentation

With this release, we provide the full Ice documentation, including a Slice API reference, as a set of online HTML pages. You can find the latest version of the Ice documentation at http://www.zeroc.com/Ice-Manual.html, and the latest version of the Slice API reference at http://www.zeroc.com/Slice-Reference.html. The documentation and reference also provide a search feature, so you can quickly locate a topic.

Of course, you can still download the Ice Manual as a PDF file, either from our download area, or by following the link in this paragraph.

Installation

This release adds binary install packages for Red Hat Enterprise Linux 4 and SUSE Linux Enterprise Server 10.

We have also added example `/etc/init.d` scripts for SUSE Linux Enterprise and Red Hat Enterprise Linux, as well as an example configuration file for Windows. This makes it easier to start Ice services as a daemon on Linux or as a service on Windows when their host machine is booted.

Build Environment

On Windows, we no longer provide Visual Studio project files. Unfortunately, these files turned out to be very brittle and created an unacceptable maintenance burden. If you want to build Ice from source, please use `nmake` instead.

Of course, there is nothing to stop you from using Visual Studio project files to build your own Ice applications—you need to use `nmake` only if you want to build Ice itself from source code. We also still ship Visual Studio project files for the demo code; you can use these project files as a starting point for creating your own project files.

Platform Support

Release 3.2 supports the following new platforms:

- Microsoft Windows Vista
- Sun Solaris 10
- SUSE Linux Enterprise Server 10
- Microsoft Windows Server 2003 (new for Ice for C# and Visual Basic)

Note that, for example, Ice for Ruby is likely to work on Windows Server 2003 too. However, we do not specifically test Ice for Ruby on this platform. (If you need explicit and guaranteed support for a particular platform, please contact us at info@zeroc.com.)

As Ice has increased in sophistication and features, so has the amount of work involved for us to create, test, document, and package each new release. As a result, we have dropped official support for a number of platforms. Of course, this does not mean that Ice does not work on these platforms (mostly likely, it will). However, we no longer explicitly test new releases on these platforms:

- Red Hat Fedora Core
- IBM AIX
- Sun Solaris 9
- Microsoft Windows 2000

We have also removed support for Microsoft .NET 1.1—Ice 3.2 C# and Visual Basic applications must use .NET 2.0.

IceGrid Replication

Benoit Foucher, Senior Software Engineer

Introduction

With the Ice 3.2 release, we have added a much asked-for feature to Ice: the replication of the IceGrid registry. With replication, it is now possible to deploy multiple registries, both to spread the load of client queries, and to provide high availability and fault tolerance. In this article, I will give you an in-depth look at IceGrid registry replication, and I will review other new features that we are providing with this release.

Replication

Presentation

The IceGrid registry uses master–slave replication: one registry runs as the master and one or more other registries run as slaves. It is possible to upgrade a slave to a master by restarting the slave.

The IceGrid registry provides a number of different services. Most of these services are replicated and accessible through the master as well as the slave registries. However, some services are accessible only through the master, while other services are accessible through slaves, but are limited in functionality. Let's review the services provided by the registry and how their functionality is affected depending on whether the registry is a master or a slave.

The most important service provided by the IceGrid registry is the location service. Similar to the DNS, it enables Ice clients to locate objects and object adapters in Ice servers without having to hard-code host names and port numbers. Like the DNS, the location service is vital to distributed applications and must be highly available; it should come as no surprise that the location service is fully replicated and provides full functionality regardless of whether the registry is a master or a slave.

Similarly, the `IceGrid::Query` interface is replicated and accessible on all registry instances. Invocations on the `IceGrid/Query` well-known object can be sent to any of the replicated registries.

The location service and the `Query` interface rely on a persistent database that contains the deployment information of IceGrid applications. A deployment can be modified only on the master registry; the master takes care of notifying the slaves of updates to the database and replicating that state in all the slaves. Whenever a slave starts up, it re-synchronizes its database with the master database (provided the master is up) to ensure that updates while the slave was down are retrieved.

The deployment information contained in the registry database is updated via the `IceGrid::Admin` interface. The IceGrid GUI and the `icegridadmin` command-line utility use this interface to deploy and manage applications. Both master and slave registries provide the `Admin` interface; however, when contacted on a slave, the `Admin` interface is limited to read-only functionality so, on a slave, you can view the deployed applications, adapters, and objects, but you cannot modify them. To make updates, you must connect to the master registry.

The administrative interface also enables you to view and shut down IceGrid registries and nodes. However, you can see the running registries only when looking at the master—slave registries are connected only to their master and are not aware of any other slaves.

Finally, the resource allocation system of the IceGrid registry is only accessible via the master. You cannot create client sessions to allocate objects via slave registries.

Here is a summary of the different features available in master and slave registries.

	Master	Slave
Query Interface	Yes	Yes
Location service	Yes	Yes
Client sessions (resource allocation system)	Yes	No
Administrative sessions (`Admin` interface)	Yes (read/write)	Yes (read-only)

Configuration

Configuring multiple instances of a registry is the same as configuring a single registry, with the addition of setting an extra property, `IceGrid.Registry.ReplicaName`. This property must be set to "Master" (or can be left unset) for the master registry; for slave registries, the property must be set to a unique replica name. Of course, the slaves must be able to contact the master registry. However, nothing special is required for this—as for all applications that use IceGrid, the `Ice.Default.Locator` property supplies a proxy that the slaves can use to locate the master.

Is that it? Yes, you really do not need to do anything else to configure the master and slave registries! (As we will see shortly, we could improve the value of the `Ice.Default.Locator` property, but let us first take a look at master and slave configuration files and then look at how to configure nodes and clients.)

Here is the configuration file of a master registry:

```
# Master Registry Configuration
IceGrid.InstanceName=DemoIceGrid
IceGrid.Registry.Client.Endpoints=default -p 12000
IceGrid.Registry.Server.Endpoints=default
IceGrid.Registry.Internal.Endpoints=default
IceGrid.Registry.Data=db/master
```

```
IceGrid.Registry.PermissionsVerifier=DemoIceGrid/
NullPermissionsVerifier
IceGrid.Registry.AdminPermissionsVerifier=DemoIceG
rid/NullPermissionsVerifier
```

And here is the configuration file of a slave registry:

```
# Slave Registry Configuration
Ice.Default.Locator=DemoIceGrid/Locator:default -p
12000
IceGrid.Registry.Client.Endpoints=default -p 12001
IceGrid.Registry.Server.Endpoints=default
IceGrid.Registry.Internal.Endpoints=default
IceGrid.Registry.Data=db/replica1
IceGrid.Registry.ReplicaName=Replica1
IceGrid.Registry.PermissionsVerifier=DemoIceGrid/
NullPermissionsVerifier
IceGrid.Registry.AdminPermissionsVerifier=DemoIceG
rid/NullPermissionsVerifier
```

Note that it is not necessary to specify the `IceGrid.InstanceName` property in the slave configuration file because the slave registry gets the instance name from the identity of the locator proxy. (The master and all slaves use the same instance name.)

The IceGrid nodes need to be connected to all active registry replicas. On start-up, a node tries to find the active registries and establishes a session with each of them. It is through this session that a registry can retrieve information on the state of the node's servers and eventually activate a server on the node. Without this session, a registry cannot obtain the endpoints of an object adapter deployed on the node and therefore may be unable to respond to locate requests. So, how does the node find all of the active registries? With the location service of course! As you can see in the sample configuration file below, the node configuration has not changed:

```
# Node Configuration
Ice.Default.Locator=DemoIceGrid/Locator:default -p
12000
IceGrid.Node.Name=node1
IceGrid.Node.Endpoints=default
IceGrid.Node.Data=db/node1
```

If you would like to see the interactions between the master and slaves, or between the nodes and registry replicas, you can enable tracing using the following properties:

- `IceGrid.Registry.Trace.Replica` controls tracing about session lifecycle between the master and slaves.
- `IceGrid.Registry.Trace.Node` and `IceGrid.Node.Trace.Replica` control tracing about session lifecycle between the nodes and registry replicas.

Now, you may wonder how to configure Ice clients to take advantage of a replicated registry. You simply specify the endpoints of each registry in the `Ice.Default.Locator` property, as shown below:

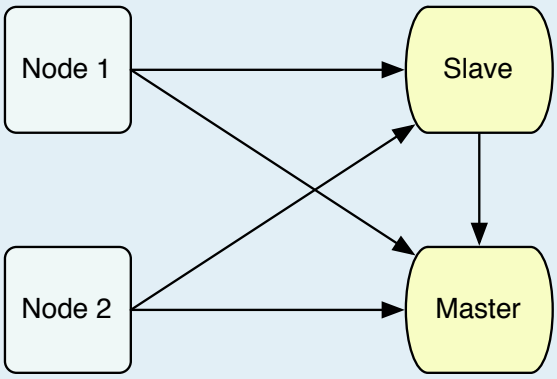
```
Ice.Default.Locator=DemoIceGrid/Locator:default -p
12000:default -p 12001
```

With this proxy, the client randomly selects a location service from the registry replicas listening on ports 12000 and 12001. If one of the replicas becomes unavailable, the client transparently connects to the other.

You might also wonder if the `Ice.Default.Locator` proxy specified in the configuration files of the slave registry and node should contain multiple endpoints. The answer is yes, but to understand the reasons we need to take a closer look at the interactions between the registries and nodes.

The diagram below illustrates the process I described earlier: when a node starts up, it tries to find all active registries and establishes a session with each of them.

Figure 1: Initial Deployment



The node contacts the registries using the default locator proxy from its configuration. If this proxy contains the endpoints of only one registry and that registry is unavailable, the node cannot find the other registries and therefore cannot connect to them. Including the endpoints of multiple registries in the locator proxy gives the node a better chance of finding an active registry.

The same is also true for slave registries. When a slave registry starts up, it uses the locator proxy to find the master registry. If the proxy contains the endpoints of only one replica and that replica is currently unavailable, the slave won't be able to contact the master registry. Although slaves do not communicate with one another, one of the slaves could be promoted to be the master at any time. In that situation, configuring each slave's locator proxy with the endpoints of all other registries provides the greatest amount of redundancy. As you can see, the best strategy is to configure the `Ice.Default.Locator` proxy for a slave, node, or client with the endpoints of all registry replicas.

Using a Replicated Registry

Now that we have learned how to properly configure a locator proxy, we can discuss other aspects of using a replicated registry.

No modifications are necessary for an application that uses the `IceGrid::Query` interface: when the Ice run time resolves the identity `IceGrid/Query`, the location service returns the endpoints of all active registry replicas.

To create an administrative session via the `IceGrid::Registry` interface, you can continue to use the `IceGrid/Registry` well-known object. This object is not replicated and is hosted by the master registry. You should use this object if you intend to create an administrative session to modify the registry's deployment information. To connect to the `IceGrid::Registry` interface of a specific replica, you can use the identity `IceGrid/Registry-<replica-name>`, where `<replica-name>` is the name of the desired replica. Remember that, if you create an administrative session with a slave registry, you are given read-only access to the deployment information.

To create client and administrative sessions through a Glacier2 router, the IceGrid master registry still provides the `IceGrid/SessionManager` and `IceGrid/AdminSessionManager` well-known objects, respectively. Slave registries provide a well-known object with the identity `IceGrid/AdminSessionManager-<replica-name>` to allow the creation of administrative sessions with the slave named `<replica-name>`. Again, an administrative session established with a slave is restricted to read-only access.

The following table summarizes the identities of the objects hosted by the registry replicas.

	Identity on the master	Identity on slave name
Locator	`IceGrid/Locator`	`IceGrid/Locator`
Query	`IceGrid/Query`	`IceGrid/Query`
Registry	`IceGrid/Registry`	`IceGrid/Registry-`*`name`*
Client session manager	`IceGrid/SessionManager`	N/A
Admin session manager	`IceGrid/AdminSessionManager`	`IceGrid/AminSessionManager-`*`name`*

Slave Promotion

You may need to promote a slave to be the new master if the current master becomes unavailable. For example, this situation can occur when the original master cannot be restarted immediately due to a hardware problem, or when your application requires a feature that is only accessible via the master, such as the resource-allocation mechanism or the ability to modify the deployment data.

To promote a slave to become the new master, you need to shut down the slave and change its `IceGrid.Registry.ReplicaName` property to "Master" (or comment out that setting). On restart, the new master notifies the nodes and registries that were active before it was shut down. An inactive registry or node will eventually connect to the new master if its default locator proxy contains the endpoint of the new master registry, or the endpoint of a slave that is connected to the new master.

If you cannot afford any down-time of the registry and want to minimize the down-time of the master, you should run at least two slaves. That way, if the master becomes unavailable, there will always be one registry available while you promote one of the slaves.

If a master registry fails and cannot be restarted for some time, you might need to update the node and registry configuration files to remove the master's endpoints from the default locator proxy. While the original master was inactive, modifications may have been recorded in the temporary master's database; restarting the original master with an obsolete database can cause an application to fail. To synchronize the original master's database, you should initially restart the original master registry as a slave. At start-up, it will synchronize its database with the temporary master registry. Then you can shut down both registries and switch the roles of master and slave.

Note that there is nothing to prevent you from running two masters. If you start two masters and these two masters contain different versions of the deployment information, some slaves and nodes might get updated with out-of-date deployment information (causing some of your servers to be deactivated). You can correct the problem by shutting down the faulty master, but it is important to keep this issue in mind when you restart a master since it might disrupt your applications.

Other new features

Standard Error, Standard Output, Log Viewing

The IceGrid administrative tools now support remote access to the following files:

- registry standard error and standard output,
- node standard error and standard output,
- server standard error and standard output, as well as user-defined log files.

To access the output of a registry or node, you must configure them to redirect to a file. The `Ice.StdErr` and `Ice.StdOut` properties define the path names of the output files. You can also define these properties for servers managed by an IceGrid node, or you can let the node generate these properties automatically for all of its servers by defining the `IceGrid.Node.Output` property to the path name of a directory in which to store the files.

For servers, you are not limited to viewing only standard error and standard output. You can also specify in a server's descriptor the path names of additional log files that administrative tools are allowed to access. Of course, if an administrative user has write-access to the deployment information, he could modify the server's

descriptor to gain access to any file on the node's host. However, with the introduction of registry replication, you can run a read-only registry by running it as a slave.

Ordered Load Balancing Policy

The replica group load balancing policy `None` is removed in this release. Instead, setting the number of replicas to 0 for any load balancing policy causes the locator to return the endpoints of all the object adapters in the replica group. (Previously, setting the number of replicas to 0 was equivalent to setting it to 1.) The load balancing policy `None` is now equivalent to the `Random` policy with the number of replicas set to 0. If your XML descriptors explicitly set the `n-replicas` attribute to 0, you should update them to set it to 1. (The registry database upgrade script `upgradeicegrid.py` automatically makes this change for deployed applications.)

We also added a new load balancing policy. The `Ordered` policy is useful when you want endpoints to be returned in order of priority. You specify priority as an integer value using the `priority` attribute of an object adapter's descriptor, as shown in the following example:

```
<application name="OrderedApp ">
  <server-template id="SimpleServer">
    <parameter name="name"/>
    <parameter name="priority"/>
    <server id="${name}" exe="./server"
     activation="on-demand">
      <adapter name="Hello"
        endpoints="tcp"
        register-process="true"
        replica-group="OrderedReplicaGroup"
        priority="${priority}"/>
      <property name="Identity" value="hello"/>
    </server>
  </server-template>
  <replica-group id="OrderedReplicaGroup">
    <load-balancing type="Ordered"
     n-replicas="2"/>
  </replica-group>
  <node name="fast-node">
    <server-instance template="SimpleServer"
     name="Main" priority="1"/>
  </node>
  <node name="slow-node">
    <server-instance template="SimpleServer"
     name="Backup" priority="2"/>
  </node>
</application>
```

When a client resolves the endpoints of the `OrderedReplicaGroup` replica group, the locator returns the endpoints of both servers such that the endpoint of `Main` appears before the endpoint of `Backup`.

For a client to use these endpoints in the proper order, you must configure its proxy with the `Ordered` endpoint selection type:

```
// C++
ObjectPrx proxy = communicator->stringToProxy(
    "hello@OrderedReplicaGroup");
proxy = proxy->ice_endpointSelection(Ordered);
proxy->ice_ping();
```

In this example, the `ice_ping` request is always sent to the `Main` server unless it is unavailable, in which case the request is sent to the `Backup` server.

Service Property Sets

Property sets were introduced in Ice 3.1 and provide a convenient way to define additional properties for server instances, especially when you want to define a property for a particular server instance without having to modify the template.

However, it was not possible to define additional properties for a specific service instance of an IceBox server instance; you could only define additional properties for the IceBox server instance and inherit these properties in all of its services. With Ice 3.2, IceBox services no longer inherit from the IceBox server properties by default, so unless you explicitly configure the IceBox server to continue to behave as in 3.1, server instance properties will not be inherited by services.

Service property sets address this issue by allowing you to configure additional properties for a specific service of an IceBox server instance. In the following example, we define the additional property `Ice.Trace.Network` for the IceStorm service of the IceStorm server instance:

```
<application name="Sample"
 import-default-templates="true">
  <node name="localhost">
    <server-instance template="IceStorm"
     name="IceStorm">
      <properties service="IceStorm">
        <property name="Ice.Trace.Network"
         value="1"/>
      </properties>
  </node>
</application>
```

Note that the IceStorm server template is imported from the registry's default templates, which are defined in the `templates.xml` file included with your Ice distribution.

Administrative changes

The `IceGrid.Registry.Admin` endpoints no longer exist in Ice 3.2. These endpoints used to provide access to the administrative interface and the Glacier2 session managers. A new (and optional) set of endpoints now supports the Glacier2 session managers: `IceGrid.Registry.SessionManager`. The administrative interface now can only be accessed in an administrative session,

which is created by the `IceGrid::Registry` interface on the endpoints defined in `IceGrid.Regisrty.Client.Endpoints`.

The `icegridadmin` utility was changed to support the creation of administrative sessions. It now accepts the following options to configure a session:

- `-u`, `--username`: login with the given user name.
- `-p`, `--password`: login with the given password.
- `-S`, `--ssl`: authenticate through SSL.

You can also specify these settings using the properties `IceGridAdmin.Username`, `IceGridAdmin.Password`, and `IceGridAdmin.AuthenticateUsingSSL`.

The `icegridadmin` utility also lets you connect to a specific registry replica using the `-R` or `--replica` option or the `IceGridAdmin.Replica` property. We also improved the `help` command to provide help by command categories, and you will find new commands to list the running registries (which are available only if you are connected to the master registry) and to get information about each registry. Finally, you can now view registry, node, and server standard error and standard output files as well as server log files.

Predefined Variable Changes

On Windows, the value of the `node.machine` variable now depends on the machine architecture. It can take the values `x86`, `x64`, or `IA64`.

Locate Requests During Server Deactivation

Previously, a client could receive `Ice::NoEndpointException` if a locate request was made for an object adapter while its server was being deactivated, or `Ice::ConnectionRefusedException` if the object adapter was deactivated but the server process had not terminated yet. The IceGrid node considered the server active as long as the process was still running.

This is no longer the case with Ice 3.2. The `wait-for-activation` server descriptor attribute has been replaced with the `server-lifetime` attribute. This new attribute retains the semantics of `wait-for-activation` and adds new semantics with respect to server deactivation. A server is still considered active when all of its registered object adapters having the `server-lifetime` attribute set to `true` are activated, but the server is now considered to be deactivated as soon as one of its object adapters having the `server-lifetime` attribute set to `true` is deactivated. In other words, a server is active only when all of its "server lifetime" object adapters are active.

As a result, as soon as one of these object adapters is deactivated, the server is now considered to be deactivated. Locate requests issued for the server's object adapters are queued instead of resulting in `Ice::NoEndpointException` or `Ice::ConnectionRefusedException`. The queued locate requests cause the server to be reactivated as soon as the process is terminated and are answered when the object adapter is activated again.

API Changes

The `getApplicationDescriptor` operation in the `IceGrid::Admin` interface has been replaced by the `getApplicationInfo` operation. This operation returns an `ApplicationInfo` structure that contains the application descriptor as well as new information about the application, such as the user that created the application, the creation time, the last user that modified the application, the modification time, and some versioning information.

The `IceGrid::RegistryObserver` interface has been split into three interfaces: `IceGrid::ApplicationObserver`, the `IceGrid::AdapterObserver`, and `IceGrid::ObjectObserver`. The new interface `IceGrid::RegistryObserver` supports the monitoring of registry replicas.

On-the-Wire Compatibility and Upgrading to 3.2

Due to numerous changes in the IceGrid interfaces for Ice 3.2, IceGrid components are not backward-compatible with previous releases. IceGrid 3.2 is not on-the-wire compatible with IceGrid 3.1, so you cannot run an IceGrid 3.2 registry with an IceGrid 3.1 node and vice versa. (Beginning with Ice 3.2, we intend to maintain on-the-wire compatibility with future releases so that it will be possible to mix IceGrid registries and nodes from different versions.) Note that this limitation only applies to compatibility between IceGrid registries and nodes—clients and servers built with previous versions of Ice can work with IceGrid 3.2 without change.

To upgrade to IceGrid 3.2, you must upgrade your IceGrid registry and nodes simultaneously. You also must upgrade the registry database with the `upgradeicegrid.py` script provided with your Ice distribution (in the `/usr/share/Ice-3.2b` directory for RPM distributions and in the `config` directory for other distributions).

Summary

A highly-available registry was the most-frequently requested feature for IceGrid. With Ice 3.2, the registry no longer represents a single point of failure and applications that rely on it gain greater reliability. If you would like to experiment with this new feature, we have provided a C++ example in the `demo/IceGrid/replication` directory.

As usual with each release, the other new features were added based on the feedback from our customers and the members of our developer forum. We look forward to hearing your opinions about this new release and your suggestions on how we can continue to improve IceGrid in future releases!

IceStorm 3.2

Matthew Newhook, Senior Software Engineer

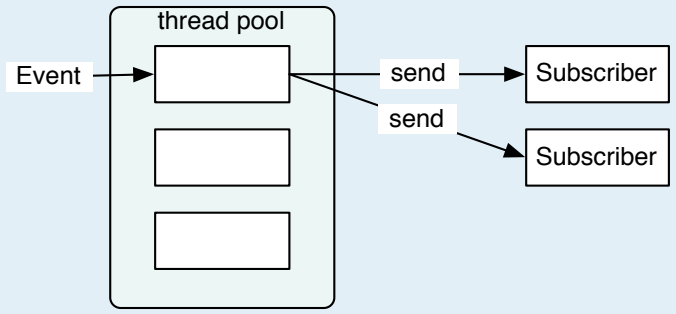
Introduction

With Ice 3.2, we have made many changes and additions to IceStorm to make the service more robust and efficient. This article provides details on these changes, as well as some background information that will help you to understand our motivation for the changes.

Subscriber Pool

In Ice 3.1, threads from the publisher thread pool forwarded events to connected subscribers. A consequence of this was that a blocked, slow, or malicious subscriber could cause an IceStorm thread to block for some time, or even forever, if no timeout was configured.

Figure 1: Ice 3.1



Note that, once all of the threads in the thread pool were consumed, a 3.1-based IceStorm could neither forward events to connected subscribers, nor process new events from publishers.

With Ice 3.2, IceStorm is significantly more resilient in this respect because it uses a separate subscriber thread pool. The threads in this pool forward events to subscribers, thereby decoupling the receiving side of IceStorm from the sending side.

To understand the need for the subscriber thread pool, we first need to look at the types of failures that caused problems in Ice 3.1. In particular, event forwarding could stop due to IceStorm's TCP/IP transport buffers filling up on the sending side or due to connection timeouts. As of release 3.2, Ice still does not guarantee that invocations will never block the caller, even for AMI and oneway invocations—see this FAQ and previous *Connections* articles for details. The subscriber thread pool is an effective way to avoid blocking despite this. Furthermore, if correctly configured, the subscriber thread pool also ensures that one misbehaved subscriber does not affect other subscribers. (Note that IceStorm 3.2 is still subject to operating system limits on the number of threads and available memory—if either of these is exhausted, IceStorm is still vulnerable to denial of service.)

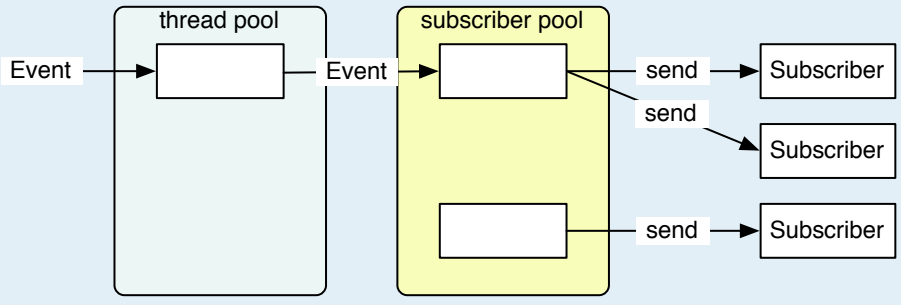
The subscriber thread pool is configured by a number of properties:

- `IceStorm.Trace.SubscriberPool`. A non-zero value enables tracing for the subscriber thread pool.
- `IceStorm.SubscriberPool.Size`. The initial size of the thread pool, with 1 as the default.
- `IceStorm.SubscriberPool.SizeMax`. The maximum size of the thread pool. The default is 0, which means no limit.
- `IceStorm.SubscriberPool.SizeWarn`. If the size of the thread pool exceeds this value, IceStorm logs a warning. The default value is 0, which disables the warning.
- `IceStorm.SubscriberPool.Timeout`. If a thread *stalls* (see below) in the preceding number of milliseconds, IceStorm adds a new thread to the subscriber thread pool (up to a limit of `IceStorm.SubscriberPool.SizeMax`). The default value is 1000 milliseconds.

Initially the pool is created with `IceStorm.SubscriberPool.Size` worker threads. These threads push events to the subscribers using the subscribers' requested quality of service (QoS).

To determine if there are problems in delivering events, IceStorm looks at the threads in the subscriber thread pool once every `IceStorm.SubscriberPool.Timeout * 10` milliseconds. A thread is deemed to be stalled if, at this time, the thread has taken longer than `IceStorm.SubscriberPool.Timeout` milliseconds to deliver an event. IceStorm adds an extra thread to the subscriber thread pool if all threads in the pool are stalled and there are pending events waiting to be delivered to subscribers. Threads are added up to the maximum specified by `IceStorm.SubscriberPool.SizeMax`. (If the number of threads exceeds `IceStorm.SubscriberPool.SizeWarn`, IceStorm also logs a warning.)

Figure 2: IceStorm SubscriberPool



If the number of stalls in the preceding `IceStorm.SubscriberPool.Timeout * 10` milliseconds is less than the number of threads above `IceStorm.SubscriberPool.Size`, IceStorm removes one of the extra threads from the pool. This means that extra threads are kept only while they are needed and that a temporary slow-down does not cause extra threads to hang around needlessly (thereby conserving resources). In effect, the subscriber pool dynamically adjusts itself to the load on the service and the speed of subscribers as appropriate.

Selection of a suitable timeout value is important. If the timeout value is too low, IceStorm will unnecessarily create extra threads. In addition, if subscribers slow down briefly but periodically, too low a timeout value can create a yoyo effect where extra threads are continuously created and destroyed. If the timeout value is too high, throughput is compromised because it takes longer for IceStorm to detect a stall. The choice of a timeout value depends on how well your application can tolerate latency of event delivery, the size of the events being sent, and the speed of the network. For most applications, the default configuration will be suitable; however, I recommend that you enable subscriber pool tracing to verify that the timeout is indeed appropriate for your application.

If IceStorm runs on a dedicated machine, with little or no load from other applications or services, I recommend that you set the publisher thread pool size to 1, and set the subscriber thread pool size to the number of CPUs of the machine. This configuration ensures optimal use of resources.

Performance

IceStorm's performance depends on the load placed on the service. Load is determined by several factors:

- the number of subscribers
- the number of publishers
- the rate of event publishing
- the size of event data

As an example, consider a sensor that publishes updates to a single topic at a rate of one event per second. If there is one subscriber to this topic, IceStorm must process one incoming invocation and one outgoing invocation per second. If the number of subscribers rises to ten, IceStorm must process one incoming invocation plus ten outgoing invocations. If we raise the number of sensors to ten as well, IceStorm now must process ten incoming invocations per second, plus one-hundred outgoing invocations (ten for each subscriber). Therefore, the total number of invocations is:

```
subscribers * publishers * publishRatePerSecond
```

The total bandwidth is

```
subscribers * publishers * publishRatePerSecond *
averageEventSize
```

You can use the throughput demo in the Ice distribution to benchmark the maximum throughput and use the latency demo to benchmark the maximum number of messages per second for your hardware and network. With these figures, you can use the preceding calculations for a back-of-the-envelope estimate of the maximum load you can impose on IceStorm. If you run IceStorm on a multi-CPU machine, you can assume that the maximum load scales more or less linearly with the number of available CPUs (provided that you are not limited by network bandwidth), so the maximum load is roughly the number of available CPUs multiplied by the number of events per second.

Delivery Method

Apart from basic load considerations, there are other factors that affect throughput. On IceStorm's receiving side, publishers control how events are sent to the service by using a twoway, oneway, datagram, batch oneway, or batch datagram proxy. In addition, IceStorm can provide any of the available transports (UDP, TCP, and SSL) on its endpoints. For benchmarking purposes, we assume that subscribers can supply events at a high enough rate to prevent IceStorm from running out of events to deliver to subscribers, so event delivery to subscribers (not event receipt from publishers) is the dominant factor that limits overall throughput.

On IceStorm's sending side, subscribers control how events are sent by IceStorm via the proxy and the quality-of-service setting (ordered or unordered) they subscribe with. The following options are possible: batch oneway, batch datagram, oneway, datagram, twoway, and twoway-ordered. For the time being, I will concentrate only on the performance aspect of these options; I will discuss their effect on delivery semantics shortly.

In terms of overall throughput, batch is fastest, followed by oneway, twoway, and twoway-ordered. Batch delivery improves overall throughput (events per second) because IceStorm combines multiple events into a single message on the wire. However, the performance improvement comes at a cost, namely increased latency because IceStorm flushes batches only at regular intervals, therefore events arrive in bursts. Moreover, if, for example, a subscriber publishes an event just after the beginning of a flush interval followed by an event just before the end of the same interval, neither event is forwarded until the interval ends.

Oneway delivery is slower than batch delivery because, typically, it requires more invocations. (IceStorm optimizes oneway delivery by forwarding events in batches if it has more than one pending event for a subscriber, instead of sending every event as a separate oneway invocation.) On the up-side, oneway delivery is less "bursty" than batch oneway delivery because it reduces overall latency.

Twoway delivery is slower than oneway delivery because it incurs return traffic on the wire. A benefit of twoway delivery is that it enables IceStorm to detect non-functional subscribers (such as non-existent subscribers and subscribers that have subscribed

to a topic with the wrong interface) and cancel their subscriptions; with oneway delivery, IceStorm cannot detect such errors and will cancel a subscription only when the connection to a subscriber becomes non-functional.

Finally, twoway-ordered delivery is slower than twoway delivery because it forces IceStorm to delay sending an event until it has received the reply for the preceding event. On the other hand, twoway-ordered delivery guarantees that events will be received by the subscriber in the same order as they were received by IceStorm. (I will return to the advantages of twoway-ordered delivery shortly.)

We extensively profiled and benchmarked IceStorm; the 3.2 version provides improved performance for all subscription types. Interestingly, the biggest gain came from forwarding oneway messages in batches and immediately flushing the batch. This approach is more efficient in memory and CPU consumption, and it also reduces lock contention and network load. (See the Ice Manual—as of this release also available as an online version—for a general discussion of batch messages.)

Configuring the Publisher Adapter

For IceStorm's publisher adapter, you can either use a thread pool or configure it to use a separate thread for each publisher. (You can set `Ice.ThreadPool.Server.Size` to increase the number of threads in the pool, or set `Ice.ThreadPerConnection` to a non-zero value to run a separate thread for each publisher.) By default, IceStorm uses a thread pool with one thread.

As a rule, you will want to leave the publisher adapter in its default configuration, at least for applications that have many more subscribers than publishers. In that case, increasing the publisher adapter's thread pool size or using a separate thread for each subscriber is unlikely to improve performance because there will be no need for IceStorm to process incoming events concurrently (but increased concurrency also increases overhead). Only if you have more publishers than subscribers might you see an improvement in throughput or latency; however, you should run benchmarks for your specific application scenario to verify any changes to the default configuration.

Note that this advice differs considerably from what is appropriate for IceStorm 3.1—when you migrate to IceStorm 3.2, you should keep this mind.

Instance Name

The property `IceStorm.InstanceName` is now used to produce unique identities for each IceStorm topic. IceStorm 3.1 used the following identities:

- *topic-name*
- *topic-name*`/publish`
- *topic-name*`/link`

In IceStorm 3.2, the corresponding identities are:

- `instance-name/topic.`*topic-name*
- `instance-name/publisher.`*topic-name*
- `instance-name/link.`*topic-name*

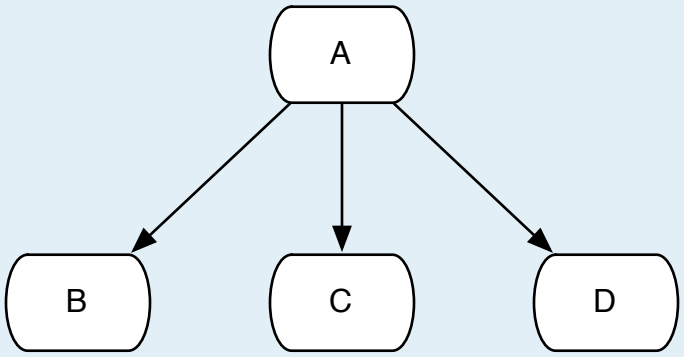
This identity format change affects the IceStorm database. We have provided a Python script, `config/upgradeicestorm.py`, that you can use to upgrade an IceStorm 3.1 database to the new format. This upgrade does not change existing identities to the new format so that everything remains fully backward-compatible.

Federation

We have made several changes to IceStorm federation to make federation of IceStorm topics both easier to use and more reliable. Firstly, what exactly is federation and why is it necessary?

Federation is a core strategy for distributing load. The basic idea is to connect an IceStorm topic to several other IceStorm topics by making several IceStorm topics a subscriber of the same topic in a different IceStorm instance.

Figure 3: Federation



As can be seen from this diagram, IceStorm topics B, C, and D are *downstream* from topic A. Conversely, topic A is *upstream* from B, C, and D. By connecting subscribers to topics B, C, and D it is possible to handle three times as many subscribers as was previously possible because each topic resides in a different IceStorm server.

This topography supports increased load in a fan-out fashion. You can also federate using a fan-in arrangement, such that a single topic subscribes to the same topic in several IceStorm servers. However, such a topography does not provide any increase in the overall maximum load because the downstream topic limits overall throughput.

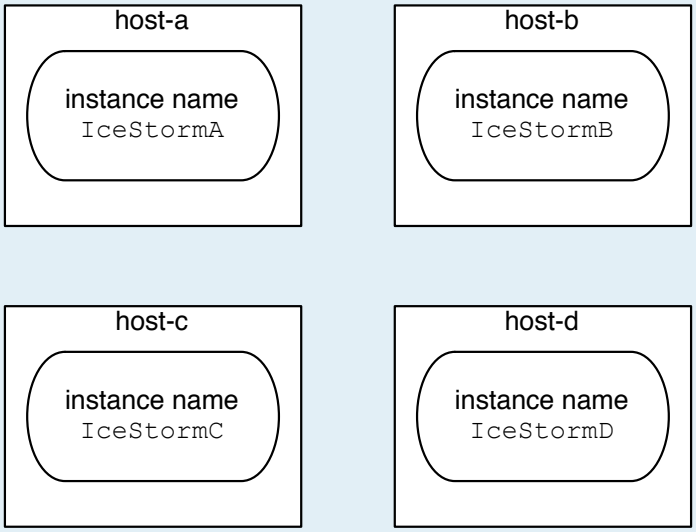
Prior to release 3.2, IceStorm ignored delivery failures to a federated topic. This meant that when a federated topic became unavailable, IceStorm continued to attempt delivery of events to the downstream topic. This caused performance problems because, in absence of the subscriber thread pool, threads in the publisher thread pool were consumed at a high rate if the failure lasted for

some time. Starting with Ice 3.2, IceStorm now detects when a downstream topic becomes unavailable. Once IceStorm diagnoses a link to a downstream topic as non-functional, it discards events for that topic. Periodically, it checks whether the downstream topic has become functional again and, if so, re-enables delivery of events to that topic. You can configure the amount of time in between attempts to re-enable delivery by setting `IceStorm.Discard.Interval`. The units are milliseconds, with a default of one minute.

We also updated `icestormadmin` to simplify the management of topic federations. Because the XML file format made it impossible to create federations of multiple IceStorm servers, we removed the `graph` command. Instead, you can use the program's `link` command to create federations. The `link` command allows you to specify a particular IceStorm server by providing the instance name of its topic manager.

You can provide an instance name with other commands as well, so you can administer any number of IceStorm servers from a single location. To enable this feature, each server must have a unique instance name. You make the instance names known to `icestormadmin` by setting the property `IceStormAdmin.TopicManager.`*`<name>`* to the proxy of the topic manager of each server. For example, suppose we have the following IceStorm deployment:

Figure 4: Initial Deployment



With this deployment, you would use the following configuration:

```
# icestormadmin configuration
IceStormAdmin.TopicManager.A=IceStormA/TopicManage
r:tcp -h host-a -p 10000
IceStormAdmin.TopicManager.B=IceStormB/TopicManage
r:tcp -h host-b -p 10000
IceStormAdmin.TopicManager.C=IceStormC/TopicManage
r:tcp -h host-c -p 10000
IceStormAdmin.TopicManager.D=IceStormD/TopicManage
r:tcp -h host-d -p 10000
```

The various `icestormadmin` commands take the topic manager as an optional part of the arguments. For example, to create a topic named `time` on topic manager `IceStormA` you would use:

```
> create IceStormA/time
```

Note that the instance name is used to identify a particular IceStorm server, not the *`<name>`* suffix in the `IceStormAdmin.TopicManager.`*`<name>`* configuration property.

`icestormadmin` provides a `current` command that sets a default topic manager. For example, to create the above `time` topic in `IceStormA`, you could also write:

```
> current IceStormA
> create time
```

The property `IceStormAdmin.TopicManager.Default` sets the initial default topic manager. (For backward-compatibility, `icestormadmin` also continues to recognize the property `IceStorm.TopicManager.Proxy`, which serves the same purpose.)

The `list` command now also accepts an instance name, allowing you to target it at a specific IceStorm instance. Creating two topics in different servers and federating them is now as easy as:

```
> create IceStormA/time
> create IceStormB/time
> link IceStormA/time IceStormB/time
```

Per-Subscriber Publisher Objects

With Ice 3.2, IceStorm now supports per-subscriber publisher objects. An event published using such an object is forwarded only to one specific subscriber. Consider a typical IceStorm application that obtains a publisher and publishes events:

```
// C++
ClockPrx clock = ClockPrx::uncheckedCast(
    topic->getPublisher());
clock->tick();
```

This code publishes a tick event to all subscribers to the topic. To publish an event to only one specific subscriber, you can use the following code:

```
// C++
ClockPrx clock = ClockPrx::uncheckedCast(
    topic->subscribeAndGetPublisher(
        IceStorm::QoS(), subscriber));
clock->tick();
```

This publishes the event only to the subscriber specified by the `subscriber` argument. To see why this is useful, consider the observer pattern. For example, we might have a list of users whose contents change over time, and an observer of the list. Without using IceStorm, a canonical interface for this would look something like the following:

```
// Slice
interface ListObserver
{
   void update();
};

interface List
{
    void attach(ListObserver* observer);
    void detatch(ListObserver* observer);
    StringSeq getContent();
};
```

A specific observer would look something like:

```
// C++
void
ListI::addUser(const string& user)
{
   // call update on each observer
}

void
ListObserverI::update()
{
    StringSeq users = _list->getContent();
    // Do whatever
}
```

Via IceStorm, the list can publish to a given topic and the list observer can subscribe and unsubscribe to that topic by calling `attach` and `detach`. Note that the above code calls back from the observer to the list to get the updated state. This is not all that scalable because, with many observers, the list must handle the load of returning the new state to all the observers. Instead, it is better to broadcast the new state over the topic:

```
// Slice
interface ListObserver
{
   void add(string user);
   void remove(string user);
};
```

This avoids the need for the callback and makes the system more scalable. However, with this design, updates to the list are broadcast incrementally, one change at a time, so the observers must be informed of the initial state of the list when they attach. Typically this is done with an `init` method:

```
// Slice
interface ListObserver
{
   void init(StringSeq users);
   // ...
};
```

`attach` is coded something like this:

```
// C++
void
ListI::attach(const ListObserverPrx& observer)
{
    observer->init(getContent());
    _topic->subscribe(IceStorm::QoS(), observer);
}
```

However, this code has a problem. What if the state changes between the call to `init` and the subscription? In this case, the observer gets an inconsistent view of the list because it misses one or more updates. (Changing the order of initialization and subscription does not help because then the observer can get updates before it gets the initial dataset.) Here is a possible solution:

```
// C++
void
ListI::attach(const ListObserverPrx& observer)
{
    IceUtil::Mutex::Lock sync(*this);
    observer->init(getContent());
    _topic->subscribe(IceStorm::QoS(), observer);
}

void
ListI::addUser(const string& user)
{
   IceUtil::Mutex::Lock sync(*this);
   ListObserverPrx pub = ListObserverPrx::
uncheckedCast(_topic->getPublisher());
   pub->addUser(user);
   _content.push_back(user);
}
```

This solves the problem with synchronization but creates its own problem because of the twoway call in the `attach` method: the entire list remains locked for the duration of the `init` call, so a misbehaved observer can block further updates to the list (potentially indefinitely). (Note that making invocations on the IceStorm service with the mutex locked is not a problem because IceStorm will not block the caller for any length of time.) There are various other ways to solve this problem, such as sending the `init` call through IceStorm itself; however, they are all somewhat complex and incur additional overhead. IceStorm makes it easy to deal with this problem through its per-subscriber publisher object:

```
// C++
void
ListI::attach(const ListObserverPrx& observer)
{
    IceUtil::Mutex::Lock sync(*this);
    ListObserverPrx subPub =
        ListObserverPrx::uncheckedCast(
            _topic->subscribeAndGetPublisher(
                IceStorm::QoS(), observer));
    subPub->init(getContent());
}

void
ListI::addUser(const string& user)
{
    IceUtil::Mutex::Lock sync(*this);
    ListObserverPrx pub =
        ListObserverPrx::uncheckedCast(
            _topic->getPublisher());
   pub->addUser(user);
   _content.push_back(user);
}
```

In this example, the `attach` method uses the per-subscriber publisher object to send the `init` call to the observer via IceStorm, instead of calling the observer directly. Because the updates received by the topic publisher and per-subscriber publisher objects are serialized, we can guarantee that the observer will get the state of the list before it receives any updates. Notice that the code still makes twoway invocations with the mutex locked; however, in contrast to the previous implementation, there is no danger of blocking because IceStorm mediates these calls.

Subscription Changes

If you call `Topic::subscribe`, you will get the same behavior as with IceStorm 3.1. However, we have deprecated `subscribe`; instead, you should call `Topic::subscribeAndGetPublisher`. This method has semantics that differ from those of `subscribe`. Firstly, if you try to subscribe a proxy with the same identity as an already-subscribed proxy, `subscribeAndGetPublisher` throws an `AlreadySubscribed` exception (instead of silently replacing the subscription, as `subscribe` does). Secondly, the allowable values of `IceStorm::QoS` differ. With `subscribe`, the `reliability` QoS could contain the values `oneway`, `batch`, `twoway`, or `twoway ordered`. With `subscribeAndGetPublisher`, the only allowable value is `ordered` (or the empty string). To select oneway, batch, or twoway subscription with IceStorm 3.2, subscribers simply pass a proxy of the appropriate type. Also note that the default subscription mode is now twoway (since that is the default mode of a proxy), whereas, previously, it was oneway.

Here are a few examples that compare IceStorm 3.1 and IceStorm 3.2 code:

```
// C++
// Old
IceStorm::QoS qos;
// Subscribe with oneway sematics
qos["reliability"] = "oneway";
topic->subscribe(qos, subscriber);
// New
topic->subscribeAndGetPublisher(
    IceStorm::QoS(), subscriber->ice_oneway());

// Old
IceStorm::QoS qos;
// Subscribe with oneway datagram sematics
qos["reliability"] = "oneway";
topic->subscribe(qos, subscriber->ice_datagram());
// New
topic->subscribeAndGetPublisher(
    IceStorm::QoS(), subscriber->ice_datagram());

// Old
IceStorm::QoS qos;
// Subscribe with oneway batch sematics
qos["reliability"] = "batch";
topic->subscribe(qos, subscriber);
// New
topic->subscribeAndGetPublisher(
    IceStorm::QoS(),
    subscriber->ice_batchOneway());

// Old
IceStorm::QoS qos;
// Subscribe with oneway batch datagram sematics
qos["reliability"] = "batch";
topic->subscribe(qos, subscriber->ice_datagram());
// New
topic->subscribeAndGetPublisher(
    IceStorm::QoS(),
    subscriber->ice_batchDatagram());

// Old
IceStorm::QoS qos;
// Subscribe with twoway sematics
qos["reliability"] = "twoway";
topic->subscribe(qos, subscriber);
// New
topic->subscribeAndGetPublisher(
    IceStorm::QoS(), subscriber->ice_twoway());

// Old
IceStorm::QoS qos;
// Subscribe with twoway ordered sematics
qos["reliability"] = "twoway ordered";
topic->subscribe(qos, subscriber);
// New
qos["reliability"] = "ordered";
topic->subscribeAndGetPublisher(
    qos, subscriber->ice_twoway());
```

Ordering Semantics

When developing an application that uses IceStorm, you need to select a QoS when publishing messages and subscribing to IceStorm. As previously discussed, the selected QoS affects performance. However, QoS also affects ordering.

What do we mean by ordering? Ordering is relevant only in the context of a single event publisher, that is, it refers to relative ordering of events that originate from a single publisher, not to absolute ordering of events that originate from several publishers. Ordering is the guarantee (or lack thereof) that a subscriber receives events from a particular publisher in the same order as the publisher sent them. In the list example we saw earlier, ordering clearly is important: an observer definitely must receive add and remove events for a user in the original order, otherwise it gets a view that is inconsistent with the real state of the list.

If you want events to be ordered, not only is the QoS important, but also the concurrency model that is used by the publisher, IceStorm, and the subscriber—doing the right thing only in the publisher is not good enough.

Publisher

On the publisher side, the simplest option to ensure ordering is to use twoway invocations to IceStorm. For example:

```
// C++
ListObserverPrx observer =
    ListObserverPrx::uncheckedCast(
        topic->getPublisher());
observer->addUser("matthew");
observer->removeUser("matthew");
```

Another option is to use oneway invocations to IceStorm, and configure an appropriate concurrency model for IceStorm, either thread-per-connection or a thread pool with a single thread. Both of these models ensure that invocations cannot be dispatched out of order. (I also recommend reading the section *The Ice Run Time in Detail—Oneway Invocations* in the Ice Manual, or this FAQ.)

Alternatively, if you only care about ordering within a group of events, you can use batch oneway. For example, you might have two groups of events A, B, C and D, E, F. In this case, you can put A, B, and C in one batch and flush, and then D, E, and F into another batch and flush. This maintains ordering within each group because messages in a batch are dispatched in order (but there is no such guarantee for messages in different batches, unless there is only one thread in the server-side thread pool, or you use thread-per-connection). Note that, starting with Ice 3.2, Ice automatically flushes batches once they reach the maximum message size (which could be quite small if UDP is being used)—if you decide to use batch events to maintain ordering, you should disable automatic flushing by setting `Ice.BatchAutoFlush` to 0.

Subscriber

For subscribers with more than one thread in their thread pool, the simplest option to maintain ordering is to use twoway-ordered QoS when subscribing. For example:

```
// C++
IceStorm::QoS qos;
qos["reliability"] = "ordered";
topic->subscribeAndGetPublisher(
    qos, subscriber->ice_twoway());
```

Note that selecting only the twoway QoS is not enough:

```
// C++. No ordering guarantee!
topic->subscribeAndGetPublisher(
    IceStorm::QoS(), subscriber->ice_twoway());
```

The key difference between twoway and twoway-ordered is that with twoway–ordered, IceStorm waits for a reply from the subscriber before publishing another event to the same subscriber. This means that, even if the subscriber uses several threads for message dispatch, events will be processed by the subscriber in the same order as they were sent. In contrast, with only twoway delivery (without the ordered QoS), IceStorm sends an event as soon as it becomes available (without waiting for a reply to the previous event), meaning that events can be dispatched out of order in the subscriber if the subscriber uses multiple threads in its server-side thread pool. With respect to ordering, twoway delivery is very similar to oneway delivery; however, oneway messages can be lost without IceStorm ever noticing, whereas twoway messages cannot. (See this FAQ for details.)

The following table summarizes the ordering semantics, assuming that dispatch uses multiple threads.

QoS	Semantics
Oneway	Unordered, fast. Can lose events in the case of server-side ACM on the subscriber.
Oneway batch	Unordered, higher throughput, but high latency. Can lose events in the case of server-side ACM on the subscriber.
Twoway	Unordered. Cannot lose events in the case of server-side ACM. Slower than oneway.
Twoway ordered	Ordered. All events strictly serialized.

Oneway Subscription Caveats

If you use oneway subscription, you should be aware of a couple of things.

With oneway subscription, IceStorm is limited in its ability to clean up dysfunctional subscriptions. If IceStorm detects a failure when it forwards an event, it automatically cancels the subscription of the offending subscriber. However, with oneway subscription, IceStorm may not detect a failure. For example, this can occur when the subscriber process dies and then subsequently restarts before more events are delivered by IceStorm. If the subscriber's endpoint uses a persistent port, but the subscriber object is not re-added to the subscriber's object adapter, IceStorm will not (and cannot) notice the failure and will continue to attempt to deliver events. For this reason, if you use oneway subscriptions, it is a good idea to use a transient port for the subscriber's object adapter. That way, IceStorm will notice that the connection no longer works and cancel the stale subscription.

Be aware that, if you use UDP to deliver events to subscribers, there is no way at all for IceStorm to detect that a subscriber has disappeared. In that case, you need to make sure that subscribers unsubscribe before they disappear, otherwise stale subscriptions will accumulate in IceStorm and eventually cause the service to bog down.

Summary

IceStorm 3.2 provides improved performance, better control over resource consumption, more reliable federation, centralized administration, publishing to single subscribers, better isolation of subscribers from failures of other subscribers, improved resilience in the face of misbehaved subscribers, and automatic adjustment to load. We hope that you will find these improvements worthwhile—as always, we welcome your feedback in our developer forum.

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: How do JVM settings affect the performance of my Ice-for-Java applications?

Ice produces many short-lived objects during request processing, so giving some thought to how the Java run-time garbage collection works is worthwhile. There are many documents available on the internet that discuss performance tuning as well as memory and garbage collection of the Java virtual machine (JVM). We'll summarize some of the relevant issues here.

There are basically two ways you can alter the JVM's behavior to improve performance. Firstly, you can influence memory use by changing how memory is allocated and organized by the JVM. Secondly, you can influence how garbage collection is performed. (A third mechanism, just-in-time (JIT) compilation, increases performance by compiling the byte-code of frequently-used methods to native code; however, JIT is enabled by default and is not tunable, so we ignore it for this discussion.) This discussion focuses on the Sun Java HotSpot JVM; other JVMs may provide similar options or additional tuning features.

First, let's consider how objects are allocated and destroyed. The HotSpot JVM implements a multi-generational garbage collector. New objects are initially allocated in a *young* generation space. After a while, an object that *survives* collections in the young generation space is moved to the *tenured* generation space. When a collection occurs on the young generation space, it is called a *minor* collection. If there isn't sufficient free memory in the tenured generation space, a *major* collection occurs. Major collections are relatively expensive as they involve all live objects.

The young generation space is optimized for objects that have short life times. If your application creates lots of objects that live for a relatively short time, you can decrease the frequency of collections in the young generation space by increasing the amount of memory allocated to it. Unfortunately, this can also increase the time it takes for the young generation collection to complete. Optimal performance will involve getting the young generation to the "right" size.

A few additional notes on the young generation space:

- If you are setting an upper bound for the Java heap (see the `-Xmx` option), you need to be careful not to set the young generation space to more than half of the upper bound. This is especially important when using the serial collector (the default garbage collector on single-CPU machines). With the serial collector, the JVM reserves enough memory in the tenured generation space to ensure that a minor collection will succeed when the young generation is full of live objects. However, if there is not enough memory available in the tenured generation space, a full collection is triggered instead.
- There are other options that affect the behavior of collections in the young generation space, such as configuring the *survivor space ratios*. We won't discuss these here. However, if you want to change these options, you need to consider how Ice works before doing so.

By default, the tenured generation is sized relative to the maximum Java heap size and the size of the young generation space. If you have somehow configured the JVM such that there insufficient tenured space, you will know because you will get `java.lang.OutOfMemoryExceptions`.

Apart from the young and tenured spaces, there is also a third space, known as the *permanent* generation space. The permanent generation space is intended for objects that live for the lifespan of the application. If your application has a large number of classes, you may want to turn on GC logging for a period of time and see if the collector collects anything in the permanent generation. If it does, you should consider increasing the size of the permanent generation space. Some useful GC logging options are:

`-verbose:gc`	Enable verbose GC logging.
`-XX:+PrintGCDetails`	Provide detailed statistics about collections.
`-XX:+PrintTenuringDistribution`	Give details about young generation survivor spaces and how many objects are promoted to the tenured space.

Here are some of the options for controlling how the JVM allocates and organizes memory (please see Sun's documentation for details):

`-Xms`	Configure the initial Java heap size.
`-Xmx`	Configure the maximum Java heap size.
`-Xmn`	Configure the maximum size for the young generation space.

`-XX:+AggressiveHeap`	Instructs the VM to analyze the current operating environment and attempt to optimize settings for memory-intensive applications. This option also enables implicit garbage collection and adaptive memory sizing. It is intended for machines with large amounts of memory and multiple CPUs.
`-XX:NewRatio`	Configure the relative size of the young generation space.
`-XX:NewSize`	Configure the initial size of the young generation space.
`-XX:MaxNewSize`	Configure the maximum size of the young generation space.

So how is this relevant to Ice? Besides application-specific issues, there are some details of how Ice is implemented that affect how the JVM will behave. For example, the size of the young generation is relevant to Ice because processing a request often involves creation of transient objects. Complex types, sequences of types, and rapid-fire requests can fill up the young generation space quickly. If a request is very complex, it may result in the young generation space filling before a single request is completed. In turn, this causes some of the transient objects to *spill over* into the tenured space. Eventually, the tenured space fills up and you end up with an expensive major collection pass. Profiling typical load scenarios will help you determine a good size for the Java heap and young generation size.

Also of relevance is Ice's per-connection buffer caching feature. Marshaling buffers live for the duration of a connection between a client and server communicator. Because these buffers can be large, they can quickly fill the young generation space and cause a collection. On the other hand, with buffer caching, the buffer objects are quickly moved to the tenured space. This is a "good thing" in that this leaves more room in the young generation space for transient objects used during marshaling and unmarshaling. However, if you create and close many connections, such as in a heavily-loaded server, the tenured space may fill more quickly, causing an expensive major collection pass. Ice 3.2 includes a new configuration property, `Ice.CacheMessageBuffers`, that permits you to disable the per-connection buffering feature, thereby allowing most of the transient request data to remain completely in the young generation space. Naturally, your Ice for Java applications (especially servers) will be happier with loads of memory.

By the way, if you have read somewhere that calling `System.gc()` is a bad idea, here is why: `System.gc()` forces a *major* collection, so all of your careful tuning goes out the window if an applications calls `System.gc()`. Fortunately, if an application does this, you can run it with `-XX:+DisableExplicitGC` to make the call a no-op.

Another area that affects performance is what the garbage collector does when it needs to reclaim memory. The following collectors are available in J2SE 5.0:

- The default (serial) collector
- Throughput collector (`-XX:+UseParallelGC`)
- Concurrent low pause collector (`-XX:+UseConcMarkSweepGC`)
- Incremental collector (`-XX:+UseTrainGC`)

The default collector is a serial collector that pauses the application during minor and major collections. If the host machine has a single CPU, the serial collector will likely be as fast or faster as the other collectors.

Pausing the entire application during garbage collection wastes one or more CPUs for the duration of a collector run so, on hosts with multiple CPUs, the throughput and concurrent low-pause collectors are worth looking at. The *throughput* collector uses the same collection mechanism as the serial collector for major collections, but implements a parallel minor collector for the young generation space. On the other hand, the *concurrent low-pause* collector attempts to perform most of the work of a major collection without interrupting your application. (Your application may still pause briefly during collection, but not as long as with the serial collector.) As with the throughput collector, the concurrent low-pause collector collects objects in the young generation space in parallel. Finally, the *incremental* collector tries to perform part of the work of a major collection each time it does a minor collection to amortize the cost of a major collection. However, the incremental collector is deprecated and will eventually be removed.

Which collector works best for your application depends on the load of the application and the host system. However, unless you have a system with multiple CPUs, tuning memory configuration rather than garbage collection is likely to yield performance gains. Let's look at the Ice throughput demo as an example.

Running on a Dell Dimension 8250 (Intel P4 with HT enabled, 1GB of RAM, with CentOS 4.4 Linux), we get the following results:

Configuration	Throughput in Mbps
byte sequence send	785
byte sequence echo	795
string sequence send	41
variable length struct send	59
fixed length struct send	103

If we specify the heap size and increase the size of the young generation using the options `-Xms250m -Xmx250m -Xmn100m`:

Configuration	Throughput in Mbps
byte sequence send	870
byte sequence echo	885
string sequence send	94
variable length struct send	104
fixed length struct send	126

So by just tweaking the memory configuration a little bit, we can get a fairly sizable increase in throughput.

This machine only has a single CPU (hyper-threading notwithstanding), so let's see what happens if we throw a parallel collector (`-XX:+UseParallelGC`) into the mix.

Configuration	Throughput in Mbps
byte sequence send	840
byte sequence echo	860
string sequence send	90
variable length struct send	100
fixed length struct send	125

The parallel garbage collection doesn't appear to help here. However, on a machine with several CPUs, you should see some benefits by enabling parallel collection.

References:

Sun Microsystems. *Tuning Garbage Collection with the Java™ Virtual Machine*. http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html.

Sun Microsystems. *Ergonomics in the Java™ Virtual Machine*. http://java.sun.com/docs/hotspot/gc5.0/ergo5.html.

Sun Microsystems. *Java 2 Platform, Standard Edition (J2SE Platform), version 1.4.2, Performance White Paper*. http://java.sun.com/j2se/1.4.2/reference/whitepapers/index.html#3.

ZeroC, Inc. 2007. *Distributed Programming with Ice*. http://www.zeroc.com/Ice-Manual.html.