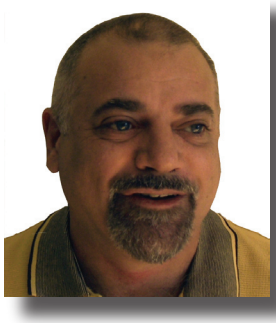
Connections

ZeroC's Newsletter for the Ice Community

Issue Number 9, December 2005



Exciting Times

I have just returned to Australia from the C++ Connections conference. This was a conference well worth attending. With a stellar lineup of speakers (including yours truly), the conference featured a raft of submissions of excellent quality. Many of the presentations mentioned the upcoming C++ 0*x* standard (with *x* likely to be 9). Among the things you can expect to make it into the standard are many of the facilities provided by the Boost library, including nice features such as better generic programming support, new data structures, and functional programming constructs. In addition—and, to me, more important than almost everything else—the standard will finally include a memory model that defines how threads interact with memory, as well as library (and, possibly, language) extensions for writing portable threaded code.

You may wonder why I am getting so excited by the addition of threading to the standard; after all, thread abstraction libraries, such as the one provided by Ice, have been around for a long time and do the job quite nicely. Of course, it's nicer to have a standard for threading than not having one, but a more important reason why threading will play a much bigger role in the future is that Moore's Law is dead in the water: CPUs will not continue to get faster at the rate we are used to. At a clock speed of 4GHz, light travels 7.5cm (or, for the metric-impaired, 3″) per clock tick. That's barely fast enough to make it from one side of the chip to the other within a clock tick. And we won't get much beyond the 4GHz speeds we have today. For one, it becomes increasingly difficult to maintain a clean clock signal at high frequencies. (Fast CPUs already devote around 30% of the chip surface to circuitry whose sole purpose is to keep the clock signal clean across the chip.) Second, heat dissipation is reaching physical limits. (The heat density inside a fast CPU is approaching the heat density inside a nuclear reactor.) And third, miniaturization won't go much further. (Quantum tunneling effects are making it increasingly difficult to reduce the size of devices much beyond what we have today.)

All this means that, for continued performance improvements, we can no longer rely on increased CPU speed to do all the hard work for us. Vendors are responding to the problem with architectures that better support parallel execution. (We recently saw the introduction of dual-core CPUs, and the trend is likely to continue to multi-core CPUs.) Increased parallelism results in better performance but, despite our best efforts, compilers are not all that good at parallelizing general-purpose application code. To realize the potential gains, we will have to write threaded code ourselves (and we will be writing a lot more of that kind of code than we do today). And this is where standardized threading will be most welcome.

Just as increased parallelism within a single machine will be important, so will be increased parallelism among different machines: the performance bottleneck will make distributed computing a much more important concern. And, of course, distribution requires middleware.

To me, this is good news. Not just because I like middleware, but because threads and distribution can take us only so far: as we are increasingly squeezed between the demand for faster applications and the hardware speed limit, we will no longer be able to afford bloatware. In this upcoming new world, simplicity, efficiency, minimalism, and elegant design and implementation will be de rigueur. Of course, these are the very principles that drive the design and implementation of Ice. And it goes to show that, long-term, technical excellence is indeed a necessary precondition for success, regardless of fashion trends in the market. (Need I mention web services here?)

I look forward to this new world, in which engineering skills will once again be more important than marketing skills. These are exciting times indeed!

Michi Henning
Chief Scientist

Issue Features

Grid Computing with IceGrid

Benoit Foucher provides an overview of the new features provided by IceGrid. IceGrid is a replacement for IcePack with many new powerful features.

Migrating from IcePack to IceGrid

In this article, Matthew Newhook explains how you can migrate your existing IcePack applications to IceGrid.

Contents

Grid Computing with IceGrid

Benoit Foucher, Senior Software Engineer

Introduction

This past month, ZeroC released Ice 3.0, which includes an exciting new service—IceGrid. IceGrid replaces IcePack and enables you to build Ice applications to be deployed on a grid network. Grid computing is the use of a network of relatively inexpensive computers to perform the computational tasks that once required costly "big iron". This article discusses the many improvements that IceGrid has over IcePack and talks a little about IceGrid's future direction.

The Past

Active Registry to Start a Node

With IcePack, an IcePack node could only start when the corresponding IcePack registry was running. This could make it inconvenient to start an IcePack node on system boot because, if the registry wasn't up when a machine was booting, you had to manually start the node.

IceGrid removes this restriction: a node can be started even if the registry isn't up. The node periodically tries to contact the registry until it succeeds. Once it succeeds, the node creates a session with the registry and periodically sends keep-alive messages. If the registry does not receive a keep-alive message in the time interval configured by the `IceGrid.Registry.NodeSessionTimeout` property, the registry considers the node dead and destroys its session. Similarly, if the registry becomes unreachable for some time, the node also destroys its session and then tries again to contact the registry to create a new session.

This new session mechanism between the node and the registry also improves response times and reliability in case a node is down: a node is only contacted if it has an active session with the registry.

Active Nodes for Deployment

To deploy an IcePack application, all the nodes participating in the application had to be running. Each node stored data about the servers it was managing and, in the event that some nodes were temporarily down, it wasn't possible to update the application.

With IceGrid, nodes no longer have an internal database; instead, all the information about the deployed application is stored in the registry database. However, the IceGrid node still maintains a directory per server in its data directory. Each server directory contains the server configuration files and the server database environments (if any). After establishing a session with the registry, the node retrieves the descriptors for the servers it manages and then synchronizes the content of the server directories accordingly. The synchronization may include activities such as updating the server configuration files, adding or removing server directories, and adding or removing database environments.

No Explicit Support for Templates

IcePack descriptors did not have explicit support for server or service templates. To emulate templates, you had to use XML includes. For example, in a `server.xml` file, you could define the following descriptor:

```
// IcePack XML descriptor
<icepack>
  <server name="${name}" kind="cpp" exe="server">
    <adapters>
      <adapter name="Hello" endpoints="default"
       register="true"/>
    </adapters>
  </server>
</icepack>
```

You then could include this server descriptor in your application descriptor:

```
// IcePack XML descriptor
<icepack>
  <application name="HelloApplication">
    <node name="localhost">
      <include descriptor="server.xml"
       name="HelloServer-1"/>
      <include descriptor="server.xml"
       name="HelloServer-2"/>
    </node>
  </application>
</icepack>
```

Although this worked fine, it wasn't possible to deploy a new server based on the server descriptor without the included XML. Also, from just reading the XML server descriptor, it was not very clear what parameters were necessary to instantiate a new server.

To remedy this, IceGrid adds explicit support for server and service templates. These template descriptors are stored with the application descriptor in the registry. This allows easy reuse of the template descriptor to instantiate new servers. Here is the IceGrid server template for the server descriptor from the example above:

```
// IceGrid XML descriptor
<icegrid>
  <application name="HelloApplication">
    <server-template id="HelloServer">
      <parameter name="instance-name"/>
      <server id="${instance-name}" exe="server">
        <adapter name="Hello" endpoints="default"
         register-process="true"/>
      </server>
    </server-template>
```

```
    <node name="localhost">
      <server-instance template="HelloServer"
       instance-name="HelloServer-1"/>
      <server-instance template="HelloServer"
       instance-name="HelloServer-2"/>
    </node>
  </application>
</icegrid>
```

As you can see, we no longer need include files to define the two server instances. The server template includes a declaration of the required parameters, which makes it easy to work out which parameters must be provided to instantiate a new server. Because server templates are stored with the registry, you can view them with the IceGrid GUI or the IceGrid admin tool:

```
$ icegridadmin --Ice.Default.Locator=...
>> server template describe HelloApplication
HelloServer
server template `HelloServer'
{
    parameters = `instance-name'
    server `${instance-name}'
    {
        exe = `server'
        activation = `manual'
        properties
        {
            Hello.Endpoints = `default'
        }
        adapter `Hello'
        {
            id = `${server}.Hello'
            endpoints = `default'
            register process = `true'
            wait for activation = `true'
        }
    }
}
```

With templates, it is trivial to deploy a new instance of the server on a new node:

```
$ icegridadmin --Ice.Default.Locator=...
>> server template instantiate HelloApplication
newnode HelloServer instance-name="HelloServer-3"
```

This instantiates a new server named `HelloServer-3`, based on the `HelloServer` template on the node `newnode`.

To experiment with server templates yourself, you can try the `demo/IceGrid/simple` demo provided with your Ice distribution. In the demo directory, you will find a descriptor named `application_with_template.xml` that makes use of templates to deploy multiple instances of the server.

Deployment Update Dependent on XML Descriptors

IcePack did not provide a way to update an existing deployment without the XML descriptors. The only way to update the deployment was to keep the XML descriptors around, edit them, and run a command to synchronize the deployment with the content of the updated XML descriptors.

With IceGrid, it is possible to edit a deployment without having the XML descriptors that were used to initially deploy the application. After transforming the XML descriptors to Slice, they are stored in the IceGrid registry database. You can programmatically retrieve these descriptors with the IceGrid admin interface, change them, and save them again in the registry. The Slice descriptors are defined in the `slice/IceGrid/Descriptor.ice` file in your Ice distribution. Of course, an easier way to update a deployment is to use the IceGrid admin GUI tool. The tool presents descriptors in graphic form and also allows you to edit them. (It is still possible to update a deployment with the XML descriptors, provided you keep them synchronized with the deployment if you make changes via the GUI.)

The Present

Replication

Prior to IceGrid, Ice provided a limited form of replication via proxies with multiple endpoints, for example: `myobject:tcp -p 10000 -h host1:tcp -p 10000 -h host2:tcp -p 10000 -h host3`. However, this form of replication is not practical in a grid environment because it requires the client to have knowledge of all the replicas and their endpoints that are deployed on the grid. The Ice run time and IceGrid now support a more elaborate form of replication: replica groups. A replica group is a virtual object adapter that is composed of multiple object adapters and can be referred to by its identifier in indirect proxies, just like a regular object adapter.

Consider the deployment shown in Figure 1. The IceGrid XML descriptor for this deployment is as follows:

```
// IceGrid XML descriptor
<icegrid>
  <application name="HelloApplication">
    <replica-group id="HelloAdapter"/>
    <server-template id="HelloServer">
      <parameter name="index"/>
      <server id="HelloServer-${index}"
       exe="server">
        <adapter name="Hello"
         id=HelloAdapter-${index}"
         replica-group="HelloAdapter"
         endpoints="tcp"/>
      </server>
    </server-template>
    <node name="host1">
      <server-instance template="HelloServer"
       index="1"/>
    </node>
    <node name="host2">
      <server-instance template="HelloServer"
       index="2"/>
    </node>
```

```
    <node name="host3">
      <server-instance template="HelloServer"
       index="3"/>
    </node>
  </application>
</icegrid>
```

The descriptor declares a replica group `HelloAdapter` and makes the `Hello` adapter a member of this group.

Figure 1 also shows the interaction between the client, the location service, and the servers. When the client invokes the `ice_ping` operation on an indirect proxy, the client-side run time first asks the locator service for the endpoints of the adapter or replica group identifier `HelloAdapter`. `HelloAdapter` refers to a replica group; depending on the load balancing policy selected for this replica group, the location service returns a set of endpoints to the client. (In this case, the locator returns all endpoints for the object adapters in the replica group because no load balancing is specified.)

You can select among different load balancing policies: random, round-robin, and adaptive. The adaptive policy selects a replica on the least loaded node. You can specify the load balancing policy with the load-balancing element in the XML descriptor, for example:

```
// IceGrid XML descriptor fragment
<replica-group id="HelloAdapter">
  <load-balancing type="round-robin"
   n-replicas="2"/>
</replica-group>
```

Here, we've selected the round robin load balancing and we choose to return the endpoints of at most two replicas to the client when it requests the replica group endpoints through the location service.

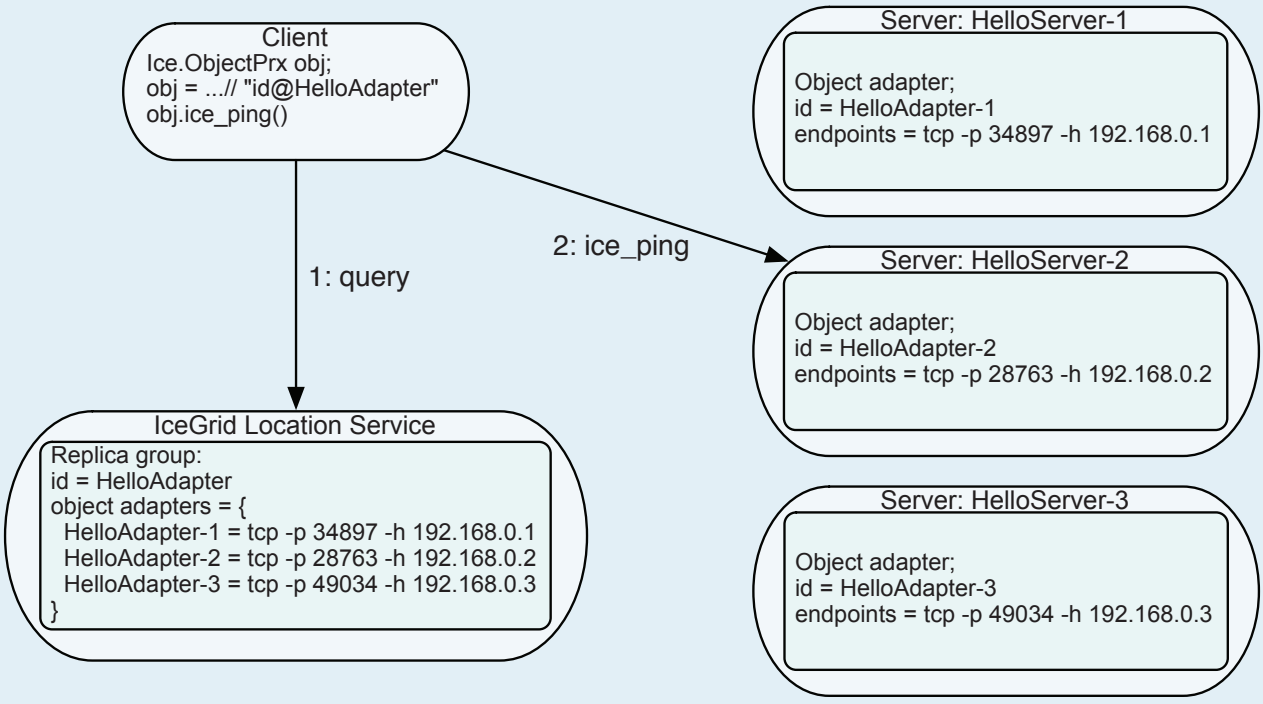
Once the client has the endpoints corresponding to the `HelloAdapter` identifier, it can invoke on the server. If the location service returns multiple endpoints, the Ice run time applies the same rules as for a direct proxy with multiple endpoints to select the endpoint to use.

Again, to experiment yourself with replication, you can use the `demo/IceGrid/simple` demo provided with your Ice distribution. In the demo directory, you'll find a descriptor named `application_with_replication.xml` which makes use of replication for the simple server.

Load monitoring and system information

An IceGrid node monitors the load of its host and sends the load information to the IceGrid registry. This allows the registry to select the replica on the least-loaded node when a client requests the endpoints of a replica group with the "adaptive" load balancing policy. The load information is also used to implement the `findObjectByTypeOnLeastLoadedNode` method on the `IceGrid::Query` interface. You can retrieve the load information of each node with the `getNodeLoad` method on the `IceGrid::Admin` interface; alternatively, you can view the load with the IceGrid admin GUI. On UNIX machines, the load is based on the load average of the machine; on Windows, the load is based on the average of the CPU utilization. In each case, the load is calculated over the previous one-, five-, or fifteen-minute period.

Figure 1: Replica Groups



Each node also provides information about its machine:

- the name of the operating system
- the release and version of the operating system
- the hostname
- the number of processors and the hardware type

This information can be retrieved via the `getNodeInfo` method on the `IceGrid::Admin` interface or viewed with the IceGrid admin GUI. The same information can also be used in deployment descriptors with the reserved variables `${node.os}`, `${node.hostname}`, `${node.release}`, `${node.version}`, and `${node.machine}`.

GUI

The IceGrid GUI allows you to browse and view deployed applications and see the status of the nodes and servers in the grid. You can also use it to create, update, or remove a deployment. When modifying a deployment, the GUI acquires an exclusive lock on the registry database to ensure the database is not concurrently modified by other processes.

To start the GUI, you can use the command

```
$ java -jar IceGridGUI.jar
```

The `IceGridGUI.jar` is located either in the `bin` or `lib` directory of the Ice distribution, depending on the platform. Once the GUI is started, it prompts for the IceGrid registry to connect to. There are two ways to connect to the registry: directly or via a Glacier2 router.

To connect directly to the registry, the GUI requires the following information:

- User name: You can specify any user name; this user name is used for IceGrid registry error messages in case there are concurrent modifications of the IceGrid registry database.
- Instance name: This name must match the value of the `IceGrid.InstanceName` property of the IceGrid registry to connect to.
- Endpoints: The endpoints much match the value of the `IceGrid.Registry.Client.Endpoints` property of the IceGrid registry to connect to.

To connect via a Glacier2 router, you must specify:

- User name and password: The user name and password used to authenticate with the Glacier2 router.
- Instance name: The instance name must match the value of the `Glacier2.InstanceName` property of the Glacier2 router to connect to.
- Endpoints: The endpoints must match the value of the `Glacier2.Client.Endpoints` property of the Glacier2 router to connect to.

Distributions

The IceGrid node has an integrated IcePatch2 client that enables it to download files for an application or server deployed on the node. This enables the download of files from an IcePatch2 server for an application or server deployed on the node.

To configure the directories to download, you need to setup a distribution. A distribution can be specified at the application descriptor level and/or the server descriptor level. An application distribution is downloaded by all the nodes for the application. A server distribution is downloaded only on the node where the server is deployed.

Let's look at an example:

```
// IceGrid XML descriptor
<icegrid>
  <application name="HelloApplication">
    <node name="localhost">
      <server id="HelloServer"
       exe="${server.distrib}/bin/server">
        <distrib
         icepatch="IcePatch2/server:tcp -p 11000">
          <directory>bin</directory>
          <directory>lib</directory>
        </distrib>
        <env>LD_LIBRARY_PATH=${server.distrib}/lib
:$LD_LIBRARY_PATH</env>
        <adapter name="Hello" endpoints="tcp"/>
      </server>
    </node>
  </application>
</icegrid>
```

In this example, we deploy a server named `HelloServer`. This server specifies a distribution that includes a `bin` and `lib` directory and specifies the IcePatch2 server proxy to use. When the application is deployed, IceGrid will download the server distribution. You can also explicitly request IceGrid to refresh the distribution on the node with the following `icegridadmin` command:

```
$ icegridadmin --Ice.Default.Locator=...
>> server patch HelloServer
```

If the server is inactive, this command will start the patch, otherwise, the patch will fail. You can either first stop the server, or use the `-f` option with the server patch command to force the patch by automatically stopping the server. Of course, you can also use the IceGrid admin GUI to initiate the patch of the server distribution.

Note that the preceding server descriptor uses another reserved variable: `${server.distrib}`. This variable can only be used in the scope of a server descriptor. Its value is the absolute path of the directory where the server distribution is downloaded by the IceGrid node.

Similarly, `${server.distrib}` is used in the definition of the `LD_LIBRRARY_PATH` environment variable. This is new with IceGrid: it is now possible to refer to environment variables in the

definition of an environment variable.

Now, let's see a descriptor containing an application distribution:

```
// IceGrid XML descriptor
<icegrid>
  <application name="HelloApplication">
    <distrib
     icepatch="IcePatch2/server:tcp -p 11000">
      <directory>data</directory>
    </distrib>
    <node name="localhost">
      <server id="HelloServer" exe="server">
        <adapter name="Hello" endpoints="tcp"/>
        <property name="DataDir"
         value="${application.distrib}/data"/>
      </server>
    </node>
  </application>
</icegrid>
```

The application distribution is specified directly in the application descriptor. All the servers for the application depend on this distribution and can access its files. As you can see, the `${application.distrib}` reserved variable specifies the application distribution directory on the node. All the servers for the application must be inactive to patch the application distribution. Furthermore, if a server depends on the application distribution, patching the server also implies patching the application distribution. If you do not want a given server to depend on the application distribution, you can set the `application-distrib` attribute to false, for example:

```
// IceGrid XML descriptor
...
      <server id="HelloServer" exe="server"
       application-distrib="false">
...
```

Distributions allow you to drop all your binaries in a central repository and get them distributed automatically to the nodes of your grid via IcePatch2. Except to install the IceGrid node binary itself, you don't need to login on the machine to deploy a new application.

Default Templates

With the Ice 3.0 distribution, you will find a number of templates for the Ice services (IceStorm, Glacier2, and IcePatch2) in the `config/templates.xml` XML descriptor. This descriptor is a special application descriptor where only templates can be defined. To use it, you need to specify its path to the IceGrid registry with the `IceGrid.Registry.DefaultTemplates` property. Once the registry is properly configured, you can import the default templates into your application descriptor. If you are using an XML descriptor to deploy your application, you need to set the `import-default-templates` attribute of the application element to true. If you use the GUI to create the application, the default behavior is to import the default templates.

Let's see how to use the IcePatch2 template in the descriptor from the previous section to deploy an IcePatch2 server with the application:

```
// IceGrid XML descriptor
<icegrid>
  <application name="HelloApplication"
   import-default-templates="true">
    <distrib>
      <directory>data</directory>
    </distrib>
    <node name="localhost">
      <service-instance template="IcePatch2"
       directory="/home/benoit/distrib"/>

      <server id="HelloServer" exe="server">
        <adapter name="Hello" endpoints="tcp"/>
        <property name="DataDir"
         value="${application.distrib}/data"/>
      </server>
    </node>
  </application>
</icegrid>
```

In this deployment, we deploy an application with a distribution and two servers: the IcePatch2 server and the `HelloServer` server. Note that we don't specify any proxy for the IcePatch2 server in the distribution; the default proxy (if none is specified) is the identity-only proxy `${application}.IcePatch2/server`. If you take a look at the IcePatch2 template in the `templates.xml` descriptor, you'll notice that this also the default identity of the IcePatch2 file server. So, by default, you do not need it to specify an identity!

Of course, you can add your own templates to this default template descriptor. This is particularly useful for service templates that are shared by multiple applications across a grid.

The Future

As usual, the addition of new major features to IceGrid will mostly depend on customer demand, so future new features are not cast in concrete. (Please don't hesitate to provide feedback on our forums!) That being said, here are a few things we plan to improve in the near future.

Descriptors

It is currently not possible to define properties for a specific server instance. You can update the server template to add the property, but then all the server instances will have this property. Or you have to copy/paste the server template and update the copy of the template. To solve this, we will add support for properties directly in server instances.

We are also planning to add support for property sets. Property sets will allow sharing a group of properties across servers and services. This is useful, for example, to define a set of debugging or tracing properties that are common to multiple servers or services.

Better integration with IcePatch2

Currently, there is no way to interrupt a patch in progress. We will add more feedback to the patching process to show the progress of a patch and to allow users to interrupt a patch. Users will also have the option to perform a thorough patch and to patch the server distribution only for a given server, even if the server depends on the application distribution.

GUI

For the GUI, we are planning to add XML import and export functionality to allow deploying applications from their XML descriptors ant to allow saving back a deployment to an XML descriptor.

Replication and Load Balancing

Currently, a client does not have much control over the selection of a replica when it invokes on a proxy that refers to a replica group. The replica to connect to is selected when the connection is established and, as long as the connection remains, the Ice run time does not try to invoke on another replica. If the connection fails, the Ice run time tries to re-establish a connection to one of the replicas. Although this behavior is fine for many applications, some applications require more advanced load balancing, such as request-based load balancing where the Ice run time selects a replica for each invocation on a proxy.

Currently, it is also not possible to control the life time of the Ice run-time locator cache entries. This is problematic for applications with long-running clients and servers that are added or removed frequently. Once the Ice run time retrieves the endpoints of a specific replica group from the location service, it caches the endpoints. The cached endpoints are kept until the communicator is destroyed or all of the endpoints become invalid (that is, connection establishment fails on all endpoints). This means that if a replica is added after the client has cached the endpoints of the replica group, the client never gets a chance to use the new replica unless all the other replicas become unreachable.

The load of the nodes is currently based on the UNIX load average or Windows CPU utilization. This has the advantage of being simple and is good enough for many applications. However, some applications need to specify what constitutes the load on a machine according to different criteria. We will add hooks to allow the developer to provide an application-specific definition of the load for a given server or node.

Summary

IceGrid provides many additional features over IcePack that make it a lot easier to deploy a large number of nodes and servers in a non-heterogeneous network environment. Thanks to IceGrid's deployment mechanism, it is possible to remotely modify, administer, and monitor a deployment. Using server templates, it is also very easy to extend a deployment. The distribution mechanism makes it trivial to distribute the binaries of an application across a grid. And, finally, replication adds scalability and fault tolerance to an application with minimal implementation effort.

As for the future of IceGrid, it all depends on you, so please don't hesitate to make suggestions on our forum!

Migrating from IcePack to IceGrid

Matthew Newhook, Senior Software Engineer

Introduction

Ice 3.0 adds an important new service to its arsenal—IceGrid. IceGrid brings grid computing to Ice. IceGrid replaces IcePack, and existing applications need to make a few changes to take advantage of the new grid functionality. This article shows you how to migrate the chat application to IceGrid as well as how to simplify the existing code and deployment. Before reading this article, I recommend that you read the "Introduction to IcePack" and "Advanced IcePack" articles in issue 5 and issue 6 of Connections, respectively.

Core Differences

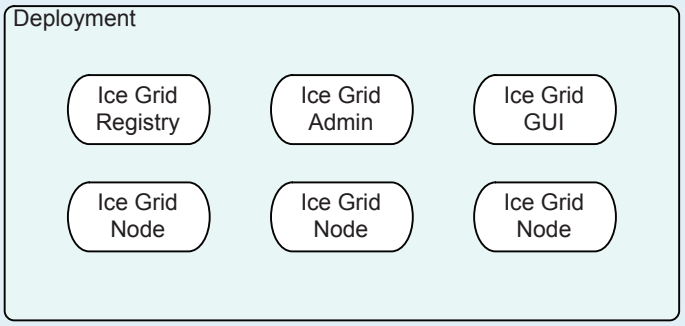
First, let's have a look at the core differences between IcePack and IceGrid. Like IcePack, IceGrid provides the locator service, server activation and monitoring service, and a server deployment mechanism. In addition, it provides

- an administration GUI
- server templates, which are a mechanism for simplifying the creation of descriptors for an application
- replication, load balancing, and automatic failover
- code, configuration, and file distribution through direct integration with IcePatch2

IceGrid consists of the components as described in Figure 1:

Compared with IcePack, the main addition is the IceGrid GUI. This application is used to monitor, create, and alter the deployment of a running grid.

Figure 1: IceGrid Components



Configuration

The configuration files remain largely the same. The same endpoint configuration as for IcePack also applies to IceGrid, with the difference that the property names use `IceGrid` instead of `IcePack`. As a first example, we'll be deploying the application on a single node, so the IceGrid configuration is as follows:

```
# IceGrid registry configuration
IceGrid.Registry.Client.Endpoints=default -p 12000
IceGrid.Registry.Server.Endpoints=default
IceGrid.Registry.Internal.Endpoints=default
IceGrid.Registry.Admin.Endpoints=default
IceGrid.Registry.Data=db/registry
#
# IceGrid node configuration.
#
IceGrid.Node.Name=localhost
IceGrid.Node.Endpoints=default
IceGrid.Node.Data=db/node
IceGrid.Node.CollocateRegistry=1
```

In addition, the configuration must define `Ice.Default.Locator`, which is used by the node to locate the registry. We also recommend that a new property `IceGrid.InstanceName` be set. This property assigns a unique object identity to an IceGrid instance. (Previously, all IcePack locators had the identity `IcePack/Locator`. Strictly speaking, this violated the Ice object model, which requires all objects to have unique identities.) The resulting configuration is as follows:

```
# IceGrid configuration (node & registry)
IceGrid.InstanceName=ChatIceGrid
# IceGrid configuration (node only)
Ice.Default.Locator=ChatIceGrid/Locator:default -p
12000
```

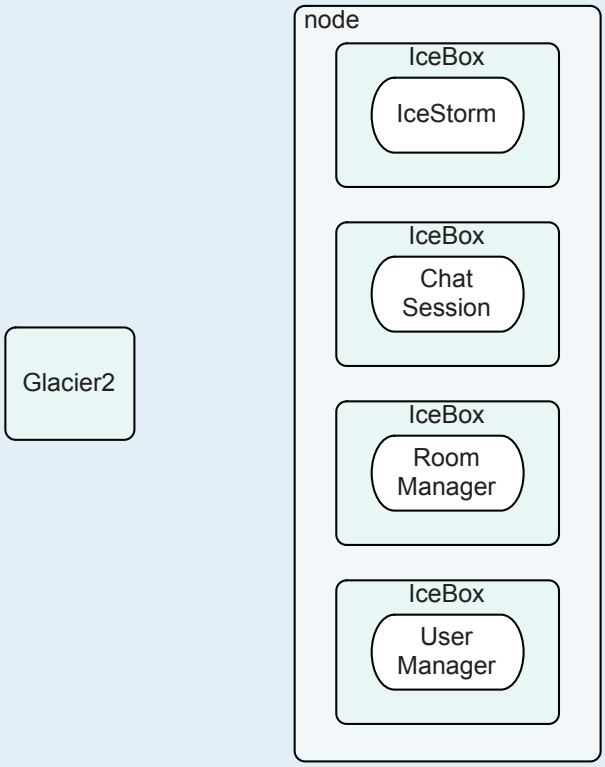
The client configuration does not change, since it receives the session object from Glacier2. The chat admin tool needs to have the locator proxy configuration, as above.

Existing Deployment

The first step is to examine the existing IcePack-based deployment. as shown in Figure 2. This deployment has one Glacier2 router, and four separate IceBox processes on a single node, with each process hosting a service. You may recall that the application uses a single room manager and user manager, but it may use many chat session and IceStorm servers. For each Glacier2 router, there is one Chat Session server.

Since we want multiple deployments of the chat session and the IceStorm servers, we can use templates. This allows us to describe the server once in a template definition, and then deploy it many times using template instances. In contrast to the equivalent IcePack construct (an `include` directive), templates are more flexible and simpler. As an example, let us examine the chat session descriptor for IcePack:

Figure 2: IcePack Deployment




```
// IcePack XML descriptor
<icepack>
  <service name="${name}"
   entry="ChatSessionService:create">
    <properties>
      <property name="Ice.ThreadPool.Client.Size"
       value="4"/>
    </properties>
    <adapters>
      <adapter name="ChatServer" endpoints="tcp">
        <object
         identity="${name}-ChatSessionManager"
         type="::Glacier2::SessionManager"/>
        <object identity="${name}-verifier"
         type="::Glacier2::PermissionsVerifier"/>
      </adapter>
    </adapters>
  </service>
</icepack>
```

This service descriptor uses the following IcePack deployment descriptor:

```
// IcePack XML descriptor
<server name="ChatSession"
 kind="cpp-icebox" endpoints="tcp -h 127.0.0.1"
 activation="on-demand">
  <include name="${server}"
   descriptor="chatsession.xml"/>
</server>
```

So what does the equivalent IceGrid template definition look like? A template definition encompasses the deployment method (`icebox` in this case) as well as the service definition. Just as the IcePack definition has a parameter, `name`, that is used to deploy multiple chat session servers, so does the IceGrid template. (Note that the `entry` parameter is not needed since it is constant for the chat session template.) The remainder of the IceGrid template is very similar to the IcePack descriptor:

```
// IceGrid XML descriptor
<server-template id="ChatSession">
  <parameter name="instance-name"/>
  <icebox id="${instance-name}" exe="icebox"
   activation="on-demand">
    <service name="${instance-name}"
     entry="ChatSessionService:create">
      <property name="Ice.ThreadPool.Client.Size"
       value="4"/>
      <adapter name="ChatServer"
       endpoints="tcp -h 127.0.0.1">
        <object identity="${instance-name}-ChatSes
sionManager"
         type="::Glacier2::SessionManager"/>
        <object identity="${instance-name}-verifie
r"
         type="::Glacier2::PermissionsVerifier"/>
      </adapter>
    </service>
  </icebox>
</server-template>
```

To deploy an instance of this server on node `localhost` using the above template definition, we use the following XML:

```
// IceGrid XML descriptor
<node name="localhost">
  <server-instance template="ChatSession"
   instance-name="ChatSession"/>
</node>
```

Note that with the above template, you cannot specify the endpoints to use as they are hard-coded directly in the template as `tcp -h 127.0.0.1`. In general, this is not a good idea since templates are supposed to be flexible and easy to deploy on a wide variety of different machines. If you were to deploy this template on a machine with multiple interfaces, the server would listen on all interfaces and any created proxies would advertise all corresponding endpoints. This is typically not what is wanted,so it is a good idea to parameterize the endpoints:

```
// IceGrid XML descriptor
<server-template id="ChatSession">
  <parameter name="instance-name"/>
  <parameter name="endpoints"
   default="tcp -h 127.0.0.1"/>
  <icebox id="${instance-name}" exe="icebox"
   activation="on-demand">
    <service name="${instance-name}"
     entry="ChatSessionService:create">
```

```
    <property name="Ice.ThreadPool.Client.Size"
     value="4"/>
    <adapter name="ChatServer"
     endpoints="${endpoints}">
```

For the IceStorm template, we make use of default templates, using the `config/templates.xml` descriptor that comes with the Ice demo distribution. (The configuration that follows assumes that the file is copied into the current working directory.) First, we set the following configuration variable in `config.icegrid`:

```
# IceGrid configuration
IceGrid.Registry.DefaultTemplates=templates.xml
```

Next, we set the `import-default-templates` attribute as follows:

```
// IceGrid XML descriptor
<application name="ChatServer"
 import-default-templates="true">
```

Now we can deploy IceStorm. (Note that we are still binding the endpoints to 127.0.0.1 since we want this deployment to run on only a single host.)

```
// IceGrid XML descriptor
<node name="localhost ">
  <server-instance template="IceStorm"
   publish-endpoints="tcp -h 127.0.0.1"
   topic-manager-endpoints="tcp -h 127.0.0.1"
   instance-name="IceStorm"/>
</node>
```

Now let's move on to the room and user managers. The IcePack descriptors were as follows:

```
// IcePack XML descriptor
<icepack>
  <service name="${name}"
   entry="RoomManagerService:create">
    <adapters>
      <adapter name="RoomManager" endpoints="tcp">
        <object identity="RoomManager"
         type="::Chat::RoomManager"/>
      </adapter>
    </adapters>
  </service>
</icepack>
<icepack>
  <service name="${name}"
   entry="Chat.UserManagerService">
    <dbenv name="${service}"/>
    <adapters>
      <adapter name="UserManager" endpoints="tcp">
        <object identity="UserManager"
         type="::Chat::UserManager"/>
      </adapter>
    </adapters>
  </service>
</icepack>
```

The corresponding IcePack deployment looked like this:

```
// IcePack XML descriptor
<server name="RoomManager" kind="cpp-icebox"
 endpoints="tcp -h 127.0.0.1"
 activation="on-demand">
  <include name="${server}"
   descriptor="roommanager.xml"/>
</server>
<server name="UserManager" kind="java-icebox"
 endpoints="tcp -h 127.0.0.1"
 activation="on-demand">
  <include name="${server}"
   descriptor="usermanager.xml"/>
</server>
```

Because the room and user managers are singletons, we need not use templates. The translation from IcePack is simple and to the point:

```
// IceGrid XML descriptor
<node name="localhost ">
  <icebox id="RoomManager" exe="icebox"
   activation="on-demand">
    <service name="RoomManager"
     entry="RoomManagerService:create">
      <adapter name="RoomManager"
       endpoints="tcp -h 127.0.0.1">
        <object identity="RoomManager"
         type="::Chat::RoomManager"/>
      </adapter>
    </service>
  </icebox>
  <icebox id="UserManager" exe="java"
   activation="on-demand">
    <option>IceBox.Server</option>
    <service name="UserManager"
     entry="Chat.UserManagerService">
      <dbenv name="${service}"/>
      <adapter name="UserManager"
       endpoints="tcp -h 127.0.0.1">
        <object identity="UserManager"
         type="::Chat::UserManager"/>
      </adapter>
    </service>
  </icebox>
</node>
```

This completes the server-side configuration. The only remaining item is Glacier2. With our IcePack-based example, we deployed Glacier2 as a manually-started component. Since IceGrid provides a default template for Glacier2 and, at present, no pass phrase is required to start Glacier2, we can set things up to have Glacier2 be managed by IceGrid. (Note that this example uses 127.0.0.1 as the client-side endpoint; in a real-world scenario, you would use the external IP address of the host.)

```
<node name="localhost">
  <server-instance template="Glacier2"
   client-endpoints="ssl -h 127.0.0.1 -p 10005"
   server-endpoints="tcp -h 127.0.0.1"
```

```
    instance-name="Glacier2"
    session-timeout="30"
    permissions-verifier="ChatSession-verifier"
    add-user-to-allow-categories="2"
    client-sleep-time="500"
    server-sleep-time="500"
    ssl-plugin="IceSSL:create"
    ssl-client-cert-path="certs"
    ssl-client-config="sslconfig.xml"
    ssl-server-cert-path="certs"
    ssl-server-config="sslconfig.xml"
  />
</node>
```

Note that the Glacier2 template does not launch Glacier2 on demand. Why is that? The reason is that the client does not use the IceGrid locator because the IceGrid locator is not (and should not be) exposed to the external network. Instead, all communications go through the Glacier2 router. Since IceGrid is not involved until after the client has contacted the Glacier2 router, it follows that IceGrid cannot automatically launch the Glacier2 router on demand.

The only other required change is to update any code references to IcePack to instead use the equivalent IceGrid APIs. This only occurs in one place—in the room manager implementation.

```
// C++
// RoomManagerI.h
const IceGrid::QueryPrx _query;

// RoomManagerI.cpp
RoomManagerI::RoomManagerI(
  const CommunicatorPtr& c) :
  _query(IceGrid::QueryPrx::checkedCast(
    c->stringToProxy("ChatIceGrid/Query")))
{
}
```

Everything else remains the same, except that the application must be linked with IceGrid instead of IcePack! The configuration and instructions for this deployment are included in the IceGrid demo distribution. I encourage you to try this out and experiment with it before we move on to more advanced topics.

Replication and Load Balancing

Assume that you want to deploy several Glacier2 routers and chat session servers. With the current model, we have one Glacier2 router for each chat session server. This one-to-one correspondence is enforced only through configuration. Here is the relevant part of the chat session descriptor template:

```
// IceGrid XML Descriptor
<adapter name="ChatServer"
 endpoints="${endpoints}">
  <object identity="${name}-ChatSessionManager"
   type="::Glacier2::SessionManager"/>
  <object identity="${name}-verifier"
   type="::Glacier2::PermissionsVerifier"/>
```

For each chat session server, we have one session manager and one permissions verifier object, and each Glacier2 router is configured with their corresponding identities. Here is the relevant part of the Glacier2 router deployment descriptor:

```
permissions-verifier="ChatSession-verifier"
session-manager="ChatSession-ChatSessionManager"
```

If we deploy a second Glacier2 router, we must also deploy a second chat session server and adjust the configuration accordingly. This not only is somewhat error prone, but also may not make the best use of server resources since a given Glacier2 router will only use a single chat session server, whether or not it is in fact the least loaded. With IceGrid we can do better.

Further, the permissions verifier object no longer belongs with the chat session manager. Given that the permissions verifier object delegates all calls to the user manager, providing multiple permissions verifier objects adds no value because each is completely dependent on a user manager object: if the user manager object is slow or unavailable, the permissions verifier object will be equally slow or unavailable. (I have omitted the code and configuration to move the verifier object; please inspect the demo for details.)

The chat session objects can be made replicas because one chat session is indistinguishable from another: a chat session does not hold externally-visible state. (As a general rule, all factory interfaces can be replicated.) IceGrid contains direct support for replication in the form of replica groups. A replica group can be thought of as a virtual object adapter that has object adapters as members. Here is a replica group descriptor for the chat session manager:

```
<replica-group id="ChatServerAdapter">
  <object identity="ChatSessionManager"
   type="::Glacier2::SessionManager"/>
</replica-group>
```

This defines a replica group with the id `ChatServerAdapter`. The group contains a well-known object `ChatSessionManager`, with the type `::Chat::ChatSession`. To support this replica group in our application, we need to alter the template for the chat session server as follows:

```
<service name="${name}"
 entry="ChatSessionService:create">
  <property name="Ice.ThreadPool.Client.Size"
   value="4"/>
  <adapter name="ChatServer"
   endpoints="${endpoints}"
   replica-group="ChatServerAdapter"/>
</service>
```

This descriptor no longer declares any well-known objects since they now come from the replica group. Now that we are using a replica group, we can change the Glacier2 router deployment as follows:

```
permissions-verifier="verifier"
session-manager="ChatSessionManager"
```

When a router first needs to use a session manager, the Ice run time first calls IceGrid to obtain all the endpoints in the replica group (because the replica group does not define a load-balancing policy), and then selects one of the endpoints that are returned at random. (See "Endpoint Filtering" in the Ice manual for more information on how the client selects an endpoint.)

For many applications, random load balancing is sufficient. However, IceGrid supports other load balancing policies as defined by a load balancing descriptor. The descriptor contains the following information:

- Type. The load balancing type can be one of random, adaptive, or round robin.
- Number of replicas. The number of endpoints to return from each query. (The default is 1.)
- Sampling interval. This attribute is only used by the adaptive load balancing policy. Each node reports its system load to the IceGrid registry at regular intervals. The sampling interval defines the time interval over which the system load average is considered. (The interval can be set to one, five, or fifteen minutes; the default value is one minute.)

Each of the load balancing policies selects one or more object adapters (defined by the configured number of replicas) and returns them to the client. The supported load balancing policies are as follows:

- Random. This load balancing policy selects a number of object adapters at random.
- Adaptive. This policy uses system load information to choose the least-loaded object adapters as defined by the sampling interval.
- Round Robin. The policy selects the least-recently-used object adapter.

As an example, the following load balancing policy for the replica group `ChatServerAdapter` selects adaptive load with a 15 minute sampling interval and returns three endpoints:

```
<replica-group id="ChatServerAdapter">
  <load-balancing type="adaptive" load-sample="15"
   n-replicas="3"/>
  <object identity="ChatSessionManager"
   type="::Glacier2::SessionManager"/>
</replica-group>
```

With Ice 3.0, we must set the property `Glacier2.SessionManager.CloseCount` to force Glacier2 to close its connection to the session manager at regular intervals and thus force it to periodically query for a new replica. This is necessary because, once it establishes a connection to a server, Ice re-uses that connection until the connection is closed. Since we haven't enabled active connection management, this means that the connection will never be closed unless the server crashes (which we hope will not occur!). Note that, even if we were to enable active connection management, the connection would be closed only if it is inactive. However, this is precisely the situation in which we don't need to use more than one session manager: if the system is inactive there is no need for multiple session managers. For this reason, you can configure Glacier2 to close its connection to the session manager on a regular basis to force a rebalance. (Future versions of Ice will allow you to control the client-side connection use in a more elegant manner.)

To force the connection to the session manager to be closed each time it is used (and thus force a reconnect each time), we can use the following configuration:

```
session-manager-close-count="1"
```

In addition, since the chat session manager no longer needs to run on the same host as the Glacier2 router, the server endpoints no longer bind to `127.0.0.1`. (In a real deployment, we would take care to ensure that the endpoints are inaccessible to the external network.) For purposes of this exercise, we leave the client-side endpoints as `127.0.0.1`. (In a real deployment, you would set these endpoints to the external interface.)

```
server-endpoints ="tcp"
```

Because the object identities of all the chat session manager objects are now the same, the chat session server code must be changed as follows:

```
class ChatSessionServiceI :
  public ::IceBox::Service
{
public:

    virtual void
    start(const string& name,
          const CommunicatorPtr& c,
          const StringSeq& args)
    {
        _adapter = c->createObjectAdapter(
          "ChatServer");
        _adapter->add(
          new ChatSessionManagerI(c),
          stringToIdentity("ChatSessionManager"));
    }
    // ...
```

Because the chat server adapter creates new session objects, we need to modify the `ChatSessionManagerI::create` implementation as follows:

```
SessionPrx
ChatSessionManagerI::create(const string& userId,
    const Current& current)
{
    Identity id = ...;
    UserPrx user = ...;
    return Glacier2::SessionPrx::uncheckedCast(
        current.adapter->add(new ChatSessionI(
        userId, user, _roomManager, _userManager),
            id));
}
```

This code actually creates a replicated proxy, that is, the Ice run time believes that this object can exist on any of the adapters in the replica group. However, this is wrong: this object should not be replicated because it belongs to one and only one chat session server. There are two ways to address this problem:

- Use a different, non-replicated, object adapter.
- Use `createDirectProxy` to return a proxy that directly points to the object hosted by the object adapter.

Since creating a new object adapter incurs some overhead, we opt for the `createDirectProxy` call. The code changes as follows:

```
SessionPrx
ChatSessionManagerI::create(const string& userId,
    const Current& current)
{
    Identity id = ...;
    UserPrx user = ...;
    current.adapter->add(
        new ChatSessionI(userId, user,
            _roomManager, _userManager), id);
    return Glacier2::SessionPrx::uncheckedCast(
        current.adapter->createDirectProxy(id));
}
```

You can use the deployment descriptor `chatapp-replicated.xml` to experiment with this deployment. Note that the endpoints no longer bind to 127.0.0.1 (except for the client side endpoint in Glacier2), since this distribution is intended to support replication and not be bound to a single host.

Distribution

After we get our application distributed on lots of nodes, we then encounter a new problem: how can we go about distributing our application executables, libraries, and data to each node? To address this, IceGrid provides an application distribution mechanism. (For the examples that follow, we assume that the Ice run time is installed on each host, and can be found in the `PATH`. For Java, we assume that `CLASSPATH` contains the Ice run time and the Berkeley DB jar files.)

For the first example, we'll get the distribution working on a single Linux host. First, we set up a directory structure to hold the distribution. We need to distribute all of the executables, as well as the certificates for the Glacier2 router. The directory setup is as follows:

- `distrib.single`: This is the root of the distribution.
- `distrib.single/certs`: This holds the contents of the `certs` subdirectory.
- `distrib.single/app`: This holds the application executables `libChatSessionService.so` and `libRoomManagerService.so`.
- `distrib.single/java`: This holds the contents of the classes subdirectory.

Once we have created this directory structure, we run `icepatch2calc` to calculate the IcePatch2 checksums:

```
$ icepatch2calc .
```

Now let's modify the deployment. First we'll deploy an IcePatch2 server using the predefined template in `templates.xml`:

```
<node name="localhost">
  <server-instance
   template="IcePatch2"
   directory="distrib.single"/>
</node>
```

Note that the directory `distrib.single` is relative to the current working directory of the IceGrid node.

Next, we need to modify the descriptors for each of our servers to add the instructions to copy the relevant distribution. Here is the relevant part of the chat session descriptor:

```
<server-template id="ChatSession">
   …
   <icebox id="${instance-name}" exe="icebox"
    activation="on-demand">
     <distrib>
       <directory>app</directory>
     </distrib>
```

The `distrib` element tells IceGrid that the files must be copied from the `app` subdirectory of the IcePatch2 distribution directory. The `icepatch` element provides the id of the IcePatch2 server to use. If you were to try this template as is, you would get an unpleasant surprise when IceGrid tries to start the chat session server: the start-up would fail because IceGrid would not be able to locate the service's shared library in the shared library path. In order to make this work, we add one final piece of magic:

```
<server-template id="ChatSession">
  ...
  <icebox id="${instance-name}" exe="icebox"
   activation="on-demand">
    <env>LD_LIBRARY_PATH=${server.distrib}/app:$LD
_LIBRARY_PATH</env>
```

The variable `${server.distrib}` contains the path name of the server's distribution directory. Note that this directive is explicit to Linux, and no longer suitable for a general-purpose template. (We'll show how to avoid this shortly.)

The room manager descriptor contains similar changes, but the user manager descriptor differs because it uses Java:

```
<icebox id="UserManager" exe="java"
 activation="on-demand">
  <env>CLASSPATH=${server.distrib}/java:$CLASSPATH
</env>
  <distrib>
    <directory>java</directory>
  </distrib>
  <option>IceBox.Server</option>
```

This is as expected: the executable class files are contained within the `java` subdirectory, which must be added to the `CLASSPATH`. How about the certificates? We could add these to either the Glacier2 template or to the application data. For this example, we add the certificates to the application data:

```
<icegrid>
  <application name="ChatServer"
   import-default-templates="true">
    <distrib>
      <directory>certs</directory>
    </distrib>
```

Now we need to change the certificate path in the Glacier2 template instantiation to use `${application.distrib}`, which is the path name to the application distribution directory:

```
ssl-plugin="IceSSL:create"
ssl-client-cert-path="${application.distrib}/certs
"
ssl-client-config="sslconfig.xml"
ssl-server-cert-path="${application.distrib}/certs
"
ssl-server-config="sslconfig.xml"
```

That's it! Please try this distribution. The full descriptor is contained in `chatapp-distrib-single.xml`.

Next, let's look at how to extend the distribution and deployment to run on multiple hosts. For this example, we'll use the deployment as described in Figure 3. Note that in a real world deployment, it is very important to ensure that the IcePatch2 server and IceGrid registry are inaccessible to the external network. This can be done by either binding the endpoints to the internal network interface, or running the services themselves on a machine that is only accessible from the internal network.

This time, we'll put the IcePatch2 distribution in a directory called `distrib`. This directory will have more or less the same contents as for the preceding example, except that it contains different executables for different operating systems. Since the executables for different operating systems may have the same name, the simplest method is to use different subdirectories, each containing the executables for a particular operating system. IceGrid provides a variable `${node.os}` that contains the operating system a given IceGrid node is running on. We'll use this variable to segregate the application executables. (Should you want to support different hardware (for example, 32-bit vs. 64-bit Linux), you can use `${node.machine}` to distinguish the operating systems.)

- `distrib/app/Linux`: This holds the Linux application executables `libChatSessionService.so` and `libRoomManagerService.so`.
- `distrib/app/Darwin`: This holds the OS-X application executables `libChatSessionService.dylib` and `libRoomManagerService.dylib`.
- `distrib/app/Windows`: This holds the application executables `ChatSessionService.dll` and `RoomManagerService.dll`.

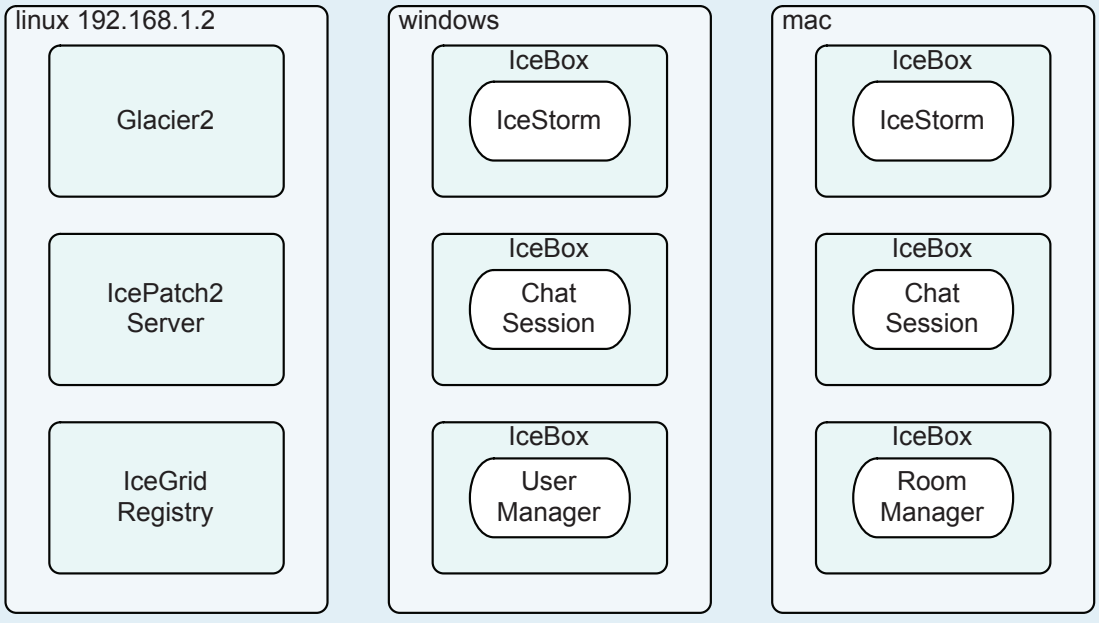
Next, we need to write the node-specific parts of the deployment file. Here is the Linux deployment, which contains the Glacier2 and the IcePatch2 server. (Again, note that the client side endpoint is specified as `127.0.0.1`; for a real-world deployment, you would use the external IP address of the server.)

```
<node name="linux">
   <server-instance
 template="Glacier2"
    client-
 endpoints="ssl -h
 127.0.0.1 -p 10005"
    server-
 endpoints="tcp"/>
   <server-instance

 template="IcePatch2"

 directory="distrib"
    endpoints="tcp"/>
</node>
```

Figure 3: Multiple Node Deployment



Next, we'll deal with the deployment configuration for the node "mac"—a Macintosh running OS-X. This machine runs the room manager server, IceStorm, and a chat session server. However, here we hit a snag: the setting of the environment variable

to control the location of shared libraries is OS-specific, so the `LD_LIBRARY_PATH` setting we used previously is inappropriate for OS-X:

```
<server-template id="ChatSession">
  <parameter name="instance-name"/>
  <icebox id="${instance-name}" exe="icebox"
   activation="on-demand">
    <env>LD_LIBRARY_PATH=${server.distrib}/app:$LD
_LIBRARY_PATH</env>
```

One solution would be to add the environment variable to the template descriptor. However, this is rather cumbersome. Since the shared library path syntax is node specific we can more easily use a node variable, like this:

```
<node name="mac">
  <variable name="shlib.path" value="DYLD_LIBRARY_
PATH=${server.distrib}/app/${node.os}:$DYLD_LIBRAR
Y_PATH"/>
```

We can reference this variable in the chat session template to set the shared library path, and we use `${node.os}` to control the distribution directory:

```
<server-template id="ChatSession">
  <parameter name="instance-name"/>
  <icebox id="${instance-name}" exe="icebox"
   activation="on-demand">
    <env>${shlib.path}</env>
    <distrib>
      <directory>app/${node.os}</directory>
    </distrib>
```

For other OS-specific variables, such as the Java `CLASSPATH`, we can also use node-specific variables as appropriate.

Onto the deployment! Because every server instance must have a unique name, the instance names of the chat session and IceStorm servers are changed accordingly.

```
<node name="mac">
  <variable name="shlib.path" value="DYLD_LIBRARY_
PATH=${server.distrib}/app/${node.os}:$DYLD_LIBRAR
Y_PATH"/>
  <server-instance template="IceStorm"
   instance-name="IceStorm-mac"/>

  <server-instance template="ChatSession"
   instance-name="ChatSession-mac"/>

  <icebox id="UserManager" exe="java"
   activation="on-demand">
    <env>CLASSPATH=${server.distrib}/java:$CLASSPA
TH</env>
    <distrib>
      <directory>java</directory>
    </distrib>
    <option>IceBox.Server</option>
    <service name="UserManager"
     entry="Chat.UserManagerService">
      <dbenv name="${service}"/>
      <adapter name="UserManager" endpoints="tcp">
        <object identity="verifier"
         type="::Glacier2::PermissionsVerifier"/>
        <object identity="UserManager"
         type="::Chat::UserManager"/>
      </adapter>
    </service>
  </icebox>
</node>
```

Here is the deployment for Windows which runs the room manager server, IceStorm, and a chat session server:

```
<node name="windows">
  <variable name="shlib.path" value="path=${server
.distrib}/app/${node.os};%path%"/>

  <server-instance template="IceStorm"
   instance-name="IceStorm-windows"/>

  <server-instance template="ChatSession"
   instance-name="ChatSession-windows"/>

  <icebox id="RoomManager" exe="icebox"
   activation="on-demand">
    <distrib>
      <directory>app/${node.os}</directory>
    </distrib>
    <service name="RoomManager"
     entry="RoomManagerService:create">
      <adapter name="RoomManager" endpoints="tcp">
        <object identity="RoomManager"
         type="::Chat::RoomManager"/>
      </adapter>
    </service>
  </icebox>
</node>
```

Before testing this deployment, we must run an IceGrid node on each host. The configuration files (`config.linux.icegrid`, `config.mac.icegrid`, `config.windows.icegrid`) can be found in the demo distribution.

That is all that is necessary to deploy on multiple operating systems. This particular deployment is not all that easy to try and experiment with because you need separate Windows, Linux, and Mac machines to try it out. However, if you have more than one machine, I encourage you to experiment with the descriptor and try out the distribution facilities in IceGrid. The deployment descriptor that I used to deploy is in `chatapp-distrib-multinode.xml`.

Conclusion

This concludes our exploration of most of IceGrid's new features. Compared to IcePack, an IceGrid deployment is both simpler and more flexible, and it makes administration across multiple nodes a much less daunting affair. At ZeroC we're very excited about IceGrid and hope you will find many uses for its powerful features.

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 can I use Java's generic types?

J2SE 5.0 (also known as JDK 1.5) introduced a generic type facility as described in JSR 14. Similar to templates in C++, a generic Java container is instantiated for a particular type; this allows the compiler to perform stricter type checking and avoids ugly casts in application code.

Introduction to Generic Types

To demonstrate the advantages of generic types, consider this traditional Java usage of a linked list container:

```
java.util.LinkedList list =
  new java.util.LinkedList();
list.add(new Integer(0));
Integer first = (Integer)list.getFirst();
int val = first.intValue();
list.add("oops"); // Allowed!
```

The `LinkedList` methods use an element type of `Object`; as a result, the compiler allows any type of object to be added to the list. If our intent is to allow only `Integer` objects in this list, then we will get no help from the compiler and it becomes our responsibility to ensure that the container holds the proper type of object. If an object of a different type does manage to become an element of this list, we'll eventually discover this fact when attempting to cast the element to `Integer` and an exception occurs. However, it would be nice to find this bug at compile time, rather than at run time, and that's where generic types really shine.

Rewriting the above example using generic types produces the following code:

```
java.util.LinkedList<Integer> list =
  new java.util.LinkedList<Integer>();
list.add(0);
int val = list.getFirst();
list.add("oops"); // Compile error!
```

There are several notable differences in this code. First, the container type is now `java.util.LinkedList<Integer>`, meaning the generic container type `java.util.LinkedList` is instantiated for integer elements. Next, it is no longer necessary to explicitly wrap an integer in an instance of the `Integer` class, as the compiler does this for you automatically. In the call to `getFirst`, for instance, not only do we avoid casting the return value to `Integer`, but we also avoid the use of `Integer` altogether. Finally, we'll get a compile-time error if we attempt to add another type of object, such as the string shown here, to our list.

Generic Types in Ice

As you can see, there are several benefits in using the new generic type facility. Naturally, this feature can only be used if you're restricting yourself to J2SE 5.0. Since Ice for Java must retain backward compatibility with J2SE 1.4, generic types are not supported by the Ice run time or by the Java code generated from the built-in Slice files (such as Glacier2, IceStorm, etc.). However, Ice for Java allows you to modify the Slice-to-Java mapping by annotating Slice definitions with metadata, and you can use this facility to generate code that supports generic types.

Metadata

The Slice language specification accepts metadata annotations on many definitions. For example, you may already be familiar with the metadata for using asynchronous method invocation and dispatch:

```
// Slice
["amd"] interface Calculator
{
    ["ami"] void longRunningOperation();
    string getName();
};
```

In this case, the interface is annotated with amd metadata, meaning all operations support asynchronous dispatch, but only `longRunningOperation` supports asynchronous invocation.

Metadata has no semantic meaning in the Slice language; compilers act on the metadata that they recognize and ignore those that they don't recognize. In addition to language-neutral metadata such as those for asynchronous invocation and dispatch, the Slice compiler for Java also accepts Java-specific metadata that modifies the mapping of sequence and dictionary types. Any alternative type can be specified, as long as it satisfies some basic requirements.

Sequences

A Slice sequence is mapped by default to a native Java array. If an alternative mapping is specified, the type must support the same methods as `java.util.List`. For example, here is how we can map a sequence of integers to a linked list using the new generic type facility:

```
// Slice
["java:type:java.util.LinkedList<Integer>"]
sequence<int> IntList;
interface I
{
  IntList getList();
};
```

When translated by the Slice-to-Java compiler, all occurrences of the `IntList` type will use `java.util.LinkedList<Integer>` instead of an int array:

```
IPrx proxy = ...
java.util.LinkedList<Integer> list =
    proxy.getList();
for(int val : list)
{
  System.out.println("entry: " + val);
}
```

Dictionaries

A Slice dictionary is mapped by default to `java.util.Map` and implemented as `java.util.HashMap`. If an alternative mapping is specified, the type must implement the `java.util.Map` interface. As an example, the following Slice definition maps a dictionary to a generic map type:

```
// Slice
["java:type:java.util.HashMap<String, Integer>"]
dictionary<string, int> StringMap;
interface I
{
  StringMap getMap();
};
```

With the help of generic types, the code to iterate over a dictionary's elements becomes easier to write and understand:

```
IPrx proxy = ...
java.util.HashMap<String, Integer> map =
    proxy.getMap();
for(String key : map.keySet())
{
  int val = map.get(key);
  System.out.println(key + " = " + val);
}
```

Q: What are Ice objects, servants, and proxies, and how do they differ?

An *Ice object* is an abstraction. Ice objects do not physically exist—there are no programming-language artifacts that would directly correspond to Ice objects, and the Ice run time does not track object existence.

The concept of an Ice object is made real by a *servant*, which is said to *incarnate* the Ice object. A servant is a programming-language object that implements the operations for an Ice object. The Ice run time sends invocations on an Ice object to the servant that incarnates the object. Note that an Ice object can exist even if no servant exists for it. For example, a server can use a servant locator to instantiate a servant for an incoming request on demand, and then destroy the servant again as soon as the request completes. In that case, the servant is destroyed, but the Ice object that was incarnated by the servant continues to exist. (Another request for the same Ice object would be sent to a new servant instance.) Note that this implies that Ice objects and servants have separate life cycles—they can be created and destroyed independently.

Each Ice object has an identity, known as the *object identity*. No two Ice objects can have the same identity. You establish the link between the conceptual Ice object and its servant when you add a servant for the Ice object to the *Active Servant Map* (*ASM*), for example, by calling `ObjectAdapter::add`. Internally, this adds an entry to a table that is used to dispatch incoming requests: the identity is the key, and a pointer or reference to the servant is the lookup value. This allows the server-side run time to track which servant should handle a request for a particular Ice object. Often, each Ice object is represented by a different servant, so there is a one-to-one correspondence between Ice objects and servants. However, you can have a single servant incarnate multiple Ice objects (for example, by using a default servant).

A *proxy* is handle that denotes a particular Ice object. You can think of a proxy as the equivalent of an object pointer, except that a proxy can denote an object in a remote address space, whereas a pointer can only point at an object in the local address space. A proxy contains the object identity of the Ice object it denotes. In addition, a proxy contains endpoint information that allows the client-side run time to find out where it can contact the server that contains the Ice object denoted by the proxy. When a client uses a proxy to invoke an operation on an Ice object, the Ice run time extracts the object identity from the proxy and sends it over the wire to the server. In turn, the server-side run time uses the object identity to locate the servant for the Ice object and, if a servant exists, dispatches the invocation to that servant.

In summary: an Ice object is a concept; a servant is the programming-artifact that implements the concept, and a proxy is a handle that allows requests to be sent to the Ice object that is incarnated by the servant.