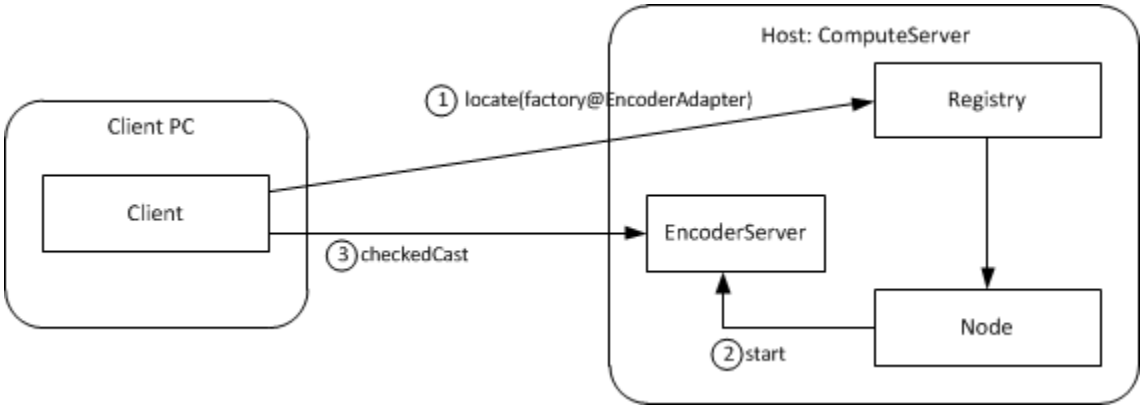
Using IceGrid Deployment

Here we extend the capabilities of our sample application using IceGrid's deployment facility.

On this page:

Ripper Architecture using Deployment

The revised architecture for our application consists of a single IceGrid node responsible for our encoding server that runs on the computer named `ComputeServer`. The illustration below shows the client's initial invocation on its indirect proxy and the actions that IceGrid takes to make this invocation possible:



Architecture for deployed ripper application.

In contrast to the initial architecture, we no longer need to manually start our server. In this revised application, the client's locate request prompts the registry to query the node about the server's state and start it if necessary. Once the server starts successfully, the locate request completes and subsequent client communication occurs directly with the server.

Ripper Deployment Descriptors

We can deploy our application using the `icegridadmin` command line utility, but first we must define our descriptors in XML. The descriptors are quite brief:

XML

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer" exe="/opt/ripper/bin/server" activation="on-demand">
                <adapter name="EncoderAdapter" id="EncoderAdapter" endpoints="tcp"/>
            </server>
        </node>
    </application>
</icegrid>
```

For IceGrid's purposes, we have named our application `Ripper`. It consists of a single server, `EncoderServer`, assigned to the node `Node1`.

> Since a computer typically runs only one node process, you might be tempted to give the node a name that identifies its host (such as `ComputeServerNode`). However, this naming convention becomes problematic as soon as you need to migrate the node to another host.

The server's `exe` attribute supplies the pathname of its executable, and the `activation` attribute indicates that the server should be activated on demand when necessary.

The object adapter's descriptor is the most interesting. As you can see, the `name` and `id` attributes both specify the value `EncoderAdapter`. The value of `name` reflects the adapter's name in the server process (i.e., the argument passed to `createObjectAdapter`) that is used for configuration purposes, whereas the value of `id` uniquely identifies the adapter within the registry and is used in indirect proxies. These attributes are not required to have the same value. Had we omitted the `id` attribute, IceGrid would have composed a unique value by combining the server name and adapter name to produce the following identifier:

```
EncoderServer.EncoderAdapter
```

The `endpoints` attribute defines one or more endpoints for the adapter. As explained earlier, these endpoints do not require a fixed port.

Refer to the XML reference for detailed information on using XML to define descriptors.

Ripper Registry and Node Configuration

In our initial registry configuration, we created the directory `/opt/ripper/registry` for use by the registry. The node also needs a subdirectory for its own purposes, so we will use `/opt/ripper/node`. Again, these directories must exist before starting the registry and node.

We also need to create an Ice configuration file to hold properties required by the registry and node. The file `/opt/ripper/config` contains the following properties:

```
# Registry properties
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsVerifier
IceGrid.Registry.Data=/opt/ripper/registry

# Node properties
IceGrid.Node.Endpoints=tcp
IceGrid.Node.Name=Node1
IceGrid.Node.Data=/opt/ripper/node
IceGrid.Node.CollocateRegistry=1
Ice.Default.Locator=IceGrid/Locator:tcp -p 4061
```

The registry and node can share this configuration file. In fact, by enabling `IceGrid.Node.CollocateRegistry`, we have indicated that the registry and node should run in the same process.

One difference from our initial configuration is that we no longer define `IceGrid.Registry.DynamicRegistration`. By omitting this property, we force the registry to reject the registration of object adapters that have not been deployed.

The node properties are explained below:

- `IceGrid.Node.Endpoints`
  This property specifies the node's endpoints. A fixed port is not required.
- `IceGrid.Node.Name`
  This property defines the unique name for this node. Its value must match the descriptor we wrote above.
- `IceGrid.Node.Data`
  This property specifies the node's data directory.
- `Ice.Default.Locator`
  This property is defined for use by the `icegridadmin` tool. The node would also require this property if the registry is not collocated. Refer to our discussion of the ripper client configuration for more information on this setting.

Ripper Server Configuration using Deployment

Server configuration is accomplished using descriptors. During deployment, the node creates a subdirectory tree for each server. Inside this tree the node creates a configuration file containing properties derived from the server's descriptors. For instance, the adapter's descriptor generates the following properties in the server's configuration file:

```
# Server configuration
Ice.Admin.ServerId=EncoderServer
Ice.Admin.Endpoints=tcp -h 127.0.0.1
Ice.ProgramName=EncoderServer
# Object adapter EncoderAdapter
EncoderAdapter.Endpoints=tcp
EncoderAdapter.AdapterId=EncoderAdapter
Ice.Default.Locator=IceGrid/Locator:default -p 4061
```

As you can see, the configuration file that IceGrid generates from the descriptor resembles the initial configuration, with two additional properties:

- `Ice.Admin.ServerId`
- `Ice.Admin.Endpoints`

These properties enable the administrative facility that, among other features, allows an IceGrid node to gracefully deactivate the server.

Using the directory structure we established for our ripper application, the configuration file for `EncoderServer` has the file name shown below:

```
/opt/ripper/node/servers/EncoderServer/config/config
```

Note that this file should not be edited directly because any changes you make are lost the next time the node regenerates the file. The correct way to add properties to the file is to include property definitions in the server's descriptor. For example, we can add the property `Ice.Trace.Network=1` by modifying the server descriptor as follows:

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer" exe="/opt/ripper/bin/server" activation="on-demand">
                <adapter name="EncoderAdapter" id="EncoderAdapter" endpoints="tcp"/>
                <property name="Ice.Trace.Network" value="1"/>
            </server>
        </node>
    </application>
</icegrid>
```

When a node activates a server, it passes the location of the server's configuration file using the `--Ice.Config` command-line argument. If you start a server manually from a command prompt, you must supply this argument yourself.

Starting the Node for the Ripper Application

Now that the configuration file is written and the directory structure is prepared, we are ready to start the IceGrid registry and node. Using a collocated registry and node, we only need to use one command:

```
$ icegridnode --Ice.Config=/opt/ripper/config
```

Additional command line options are supported, including those that allow the node to run as a Windows service or Unix daemon.

Deploying the Ripper Application

With the registry up and running, it is now time to deploy our application. Like our client, the `icegridadmin` utility also requires a definition for the `Ice.Default.Locator` property. We can start the utility with the following command:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
```

After confirming that it can contact the registry, `icegridadmin` provides a command prompt at which we deploy our application. Assuming our descriptor is stored in `/opt/ripper/app.xml`, the deployment command is shown below:

```
>>> application add "/opt/ripper/app.xml"
```

Next, confirm that the application has been deployed:

```
>>> application list
Ripper
```

You can start the server using this command:

```
>>> server start EncoderServer
```

Finally, you can retrieve the current endpoints of the object adapter:

```
>>> adapter endpoints EncoderAdapter
```

If you want to experiment further using `icegridadmin`, issue the `help` command and review the available commands.

Ripper Progress Review

We have deployed our first IceGrid application, but you might be questioning whether it was worth the effort. Even at this early stage, we have already gained several benefits:

- We no longer need to manually start the encoder server before starting the client, because the IceGrid node automatically starts it if it is not active at the time a client needs it. If the server happens to terminate for any reason, such as an IceGrid administrative action or a server programming error, the node restarts it without intervention on our part.
- We can manage the application remotely using one of the IceGrid administration tools. The ability to remotely modify applications, start and stop servers, and inspect every aspect of your configuration is a significant advantage.

Admittedly, we have not made much progress yet in our stated goal of improving the performance of the ripper over alternative solutions that are restricted to running on a single computer. Our client now has the ability to easily delegate the encoding task to a server running on another computer, but we have not achieved the parallelism that we really need. For example, if the client created a number of encoders and used them simultaneously from multiple threads, the encoding performance might actually be *worse* than simply encoding the data directly in the client, as the remote computer would likely slow to a crawl while attempting to task-switch among a number of processor-intensive tasks.

Adding Nodes to the Ripper Application

Adding more nodes to our environment would allow us to distribute the encoding load to more compute servers. Using the techniques we have learned so far, let us investigate the impact that adding a node would have on our descriptors, configuration, and client application.

Descriptor Changes

The addition of a node is mainly an exercise in cut and paste:

XML

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer1" exe="/opt/ripper/bin/server" activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp"/>
            </server>
        </node>
        <node name="Node2">
            <server id="EncoderServer2" exe="/opt/ripper/bin/server" activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp"/>
            </server>
        </node>
    </application>
</icegrid>
```

Note that we now have two `node` elements instead of a single one. You might be tempted to simply use the host name as the node name. However, in general, that is not a good idea. For example, you may want to run several IceGrid nodes on a single machine (for example, for testing). Similarly, you may have to rename a host at some point, or need to migrate a node to a different host. But, unless you also rename the node, that leads to the situation where you have a node with the name of a (possibly obsolete) host when the node in fact is not running on that host. Obviously, this makes for a confusing configuration — it is better to use abstract node names, such as `Node1`.

Aside from the new `node` element, notice that the server identifiers must be unique. The adapter name, however, can remain as `EncoderAdapter` because this name is used only for local purposes within the server process. In fact, using a different name for each adapter would actually complicate the server implementation, since it would somehow need to discover the name it should use when creating the adapter.

We have also removed the `id` attribute from our adapter descriptors; the default values supplied by IceGrid are sufficient for our purposes.

Configuration Changes

We can continue to use the configuration file we created earlier for our combined registry-node process. We need a separate configuration file for `Node2`, primarily to define a different value for the property `IceGrid.Node.Name`. However, we also cannot have two nodes configured with `IceGrid.Node.CollocateRegistry` because only one master registry is allowed, so we must remove this property:

```
IceGrid.Node.Endpoints=tcp
IceGrid.Node.Name=Node2
IceGrid.Node.Data=/opt/ripper/node

Ice.Default.Locator=IceGrid/Locator:tcp -h registryhost -p 4061
```

We assume that `/opt/ripper/node` refers to a local file system directory on the computer hosting `Node2`, and not a shared volume, because two nodes must not share the same data directory.

We have also modified the locator proxy to include the address of the host on which the registry is running.

Redeploying the Application

After saving the new descriptors, you need to redeploy the application. Using `icegridadmin`, issue the following command:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> application update "/opt/ripper/app.xml"
```

If an update affects any of the application's servers that are currently running, IceGrid automatically stops those servers prior to performing the update and restarts them again after the update is complete. We can determine whether an update would require any restarts using the `application diff` command:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> application diff --servers "/opt/ripper/app.xml"
```

To ensure that our update does not impact any active servers, we can use the `--no-restart` option:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> application update --no-restart "/opt/ripper/app.xml"
```

With this option, the update would fail if any servers required a restart.

Client Changes

We have added a new node, but we still need to modify our client to take advantage of it. As it stands now, our client can delegate an encoding task to one of the two `MP3EncoderFactory` objects. The client selects a factory by using the appropriate indirect proxy:

- `factory@EncoderServer1.EncoderAdapter`
- `factory@EncoderServer2.EncoderAdapter`

In order to distribute the tasks among both factories, the client could use a random number generator to decide which factory receives the next task:

C++

```
string adapter;
if ((rand() % 2) == 0)
    adapter = "EncoderServer1.EncoderAdapter";
else
    adapter = "EncoderServer2.EncoderAdapter";
Ice::ObjectPrx proxy = communicator->stringToProxy("factory@" + adapter);
Ripper::MP3EncoderFactoryPrx factory = Ripper::MP3EncoderFactoryPrx::checkedCast(proxy);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

There are a few disadvantages in this design:

- The client application must be modified each time a new compute server is added or removed because it knows all of the adapter identifiers.
- The client cannot distribute the load intelligently; it is just as likely to assign a task to a heavily-loaded computer as it is an idle one.

We describe better solutions in the sections that follow.

See Also

- IceGrid Server Activation
- Creating an Object Adapter
- Object Adapter Endpoints
- Getting Started with IceGrid
- icegridadmin Command Line Tool
- IceGrid and the Administrative Facility
- icegridnode
- Adapter Descriptor Element
- IceGrid Properties