# Getting Started with IceGrid

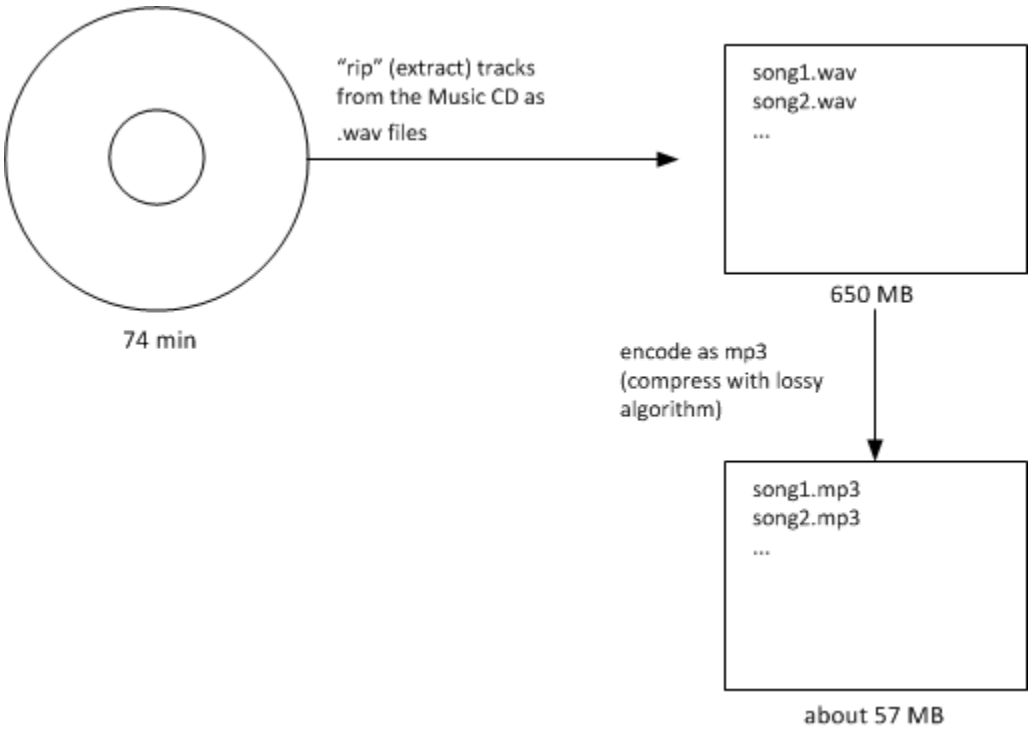
This page introduces a sample application that will help us demonstrate IceGrid's capabilities.

On this page:


- The Ripper Application
- Initial Ripper Architecture
- Ripper Registry Configuration
- Ripper Client Configuration
- Ripper Server Configuration
- Starting the Registry for the Ripper Application
- Starting the Ripper Server
- Ripper Progress Review


## The Ripper Application

Our application "rips" music tracks from a compact disc (CD) and encodes them as MP3 files, as shown below:




*Overview of sample application.*

Ripping an entire CD usually takes several minutes because the MP3 encoding requires lots of CPU cycles. Our distributed ripper application accelerates this process by taking advantage of powerful CPUs on remote Ice servers, enabling us to process many songs in parallel.

The Slice interface for the MP3 encoder is straightforward:


Copyright © 2017, ZeroC, Inc.

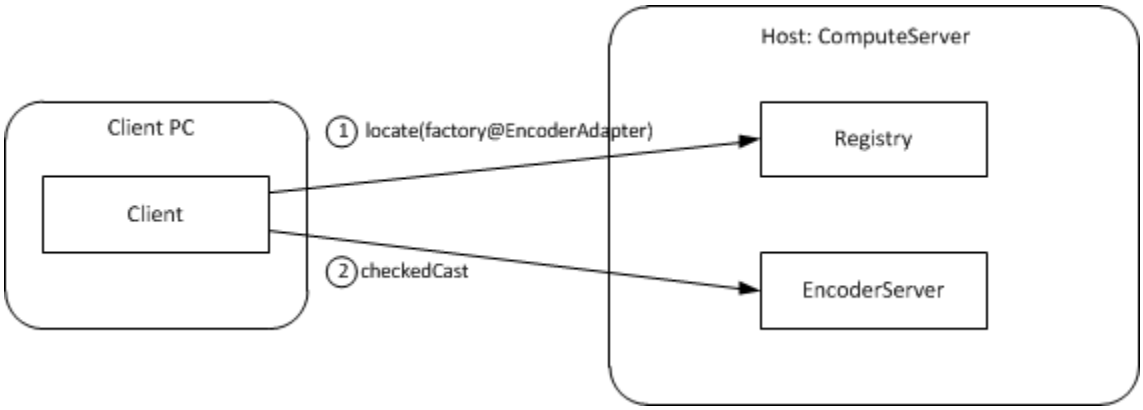
**Slice**

```
#include <Ice/BuiltinSequences.ice>
module Ripper {
exception EncodingFailedException {
    string reason;
};

sequence<short> Samples;

interface Mp3Encoder {
    // Input: PCM samples for left and right channels
    // Output: MP3 frame(s).
    Ice::ByteSeq encode(Samples leftSamples, Samples rightSamples)
        throws EncodingFailedException;

    // You must flush to get the last frame(s). Flush also
    // destroys the encoder object.
    Ice::ByteSeq flush()
        throws EncodingFailedException;
};

interface Mp3EncoderFactory
{
    Mp3Encoder* createEncoder();
};
};
```

The implementation of the encoding algorithm is not relevant for the purposes of this discussion. Instead, we will focus on incrementally improving the application as we discuss IceGrid features.

# Initial Ripper Architecture

The initial architecture for our application is intentionally simple, consisting of an IceGrid registry and a server that we start manually. This illustration shows how the client's invocation on its `EncoderFactory` proxy causes an implicit locate request:




*Initial architecture for the ripper application.*

The corresponding C++ code for the client is presented below:

**C++**

```
Ice::ObjectPrx proxy = communicator->stringToProxy("factory@EncoderAdapter");
Ripper::MP3EncoderFactoryPrx factory = Ripper::MP3EncoderFactoryPrx::checkedCast(proxy);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```


Copyright © 2017, ZeroC, Inc.

Notice that the client uses an indirect proxy for the `MP3EncoderFactory` object. This stringified proxy can be read literally as "the object with identity `factory` in the object adapter identified as `EncoderAdapter`." The encoding server creates this object adapter and ensures that the object adapter uses this identifier. Since each object adapter must be uniquely identified, the registry can easily determine the server that created the adapter and return an appropriate endpoint to the client.

The client's call to `checkedCast` is the first remote invocation on the factory object, and therefore the locate request is performed during the completion of this invocation. The subsequent call to `createEncoder` is sent directly to the server without further involvement by IceGrid.

# Ripper Registry Configuration

The registry needs a subdirectory in which to create its databases, and we will use `/opt/ripper/registry` for this purpose (the directory must exist before starting the registry). We also need to create an Ice configuration file to hold properties required by the registry. The file `/opt/ripper/registry.cfg` contains the following 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
IceGrid.Registry.DynamicRegistration=1
```

Several of the properties define endpoints, but only the value of `IceGrid.Registry.Client.Endpoints` needs a fixed port. This property specifies the endpoints of the IceGrid locator service; IceGrid clients must include these endpoints in their definition of `Ice.Default.Locator`, as discussed in the next section. The TCP port number (4061) used in this example has been reserved by the Internet Assigned Numbers Authority (IANA) for the IceGrid registry, along with SSL port number 4062.

Several other properties are worth mentioning:

- `IceGrid.Registry.AdminPermissionsVerifier`
  Controls access to the registry's administrative functionality.
- `IceGrid.Registry.Data`
  Specifies the registry's database directory.
- `IceGrid.Registry.DynamicRegistration`
  If set to a non-zero value, allows servers to register their object adapters. Dynamic registration is explained in more detail below.

By default, IceGrid will not permit a server to register its object adapters without using IceGrid's deployment facility. In some situations, such as in this sample application, you may want a client to be able to bind indirectly to a server without having to first deploy the server. That is, simply starting the server should be sufficient to make the server register itself with IceGrid and be reachable from clients.

You can achieve this by running the registry with the property `IceGrid.Registry.DynamicRegistration` set to a non-zero value. With this setting, IceGrid permits an adapter to register itself upon activation even if it has not been previously deployed. To force the server to register its adapters, you must define `Ice.Default.Locator` (so the server can find the registry) and, for each adapter that you wish to register, you must set `<adapter-name>.AdapterId` to an identifier that is unique within the registry. Setting the `<adapter-name>.AdapterId` property also causes the adapter to no longer create direct proxies but rather to create indirect proxies that clients must resolve via the registry.

# Ripper Client Configuration

The client requires only minimal configuration, namely a value for the property `Ice.Default.Locator`. This property supplies the Ice run time with the proxy for the locator service. In IceGrid, the locator service is implemented by the registry, and the locator object is available on the registry's client endpoints. The property `IceGrid.Registry.Client.Endpoints` defined above provides most of the information we need to construct the proxy. The missing piece is the identity of the locator object, which defaults to `IceGrid/Locator`:

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

The use of a locator service allows the client to take advantage of indirect binding and avoid static dependencies on server endpoints. However, the locator proxy must have a fixed port, otherwise the client has a bootstrapping problem: it cannot resolve indirect proxies without knowing the endpoints of the locator service.

Note that the identity of the locator object may change based on the registry's configuration.


Copyright © 2017, ZeroC, Inc.

# Ripper Server Configuration

We use `/opt/ripper/server.cfg` as the server's configuration file. It contains the following properties:

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

The properties are described below:

- `EncoderAdapter.AdapterId`
  This property supplies the object adapter identifier that the client uses in its indirect proxy (e.g., `factory@EncoderAdapter`).
- `EncoderAdapter.Endpoints`
  This property defines the object adapter's endpoint. Notice that the value does not contain any port information, meaning that the adapter uses a system-assigned port. Without IceGrid, the use of a system-assigned port would pose a significant problem: how would a client create a direct proxy if the adapter's port could change every time the server is restarted? IceGrid solves this problem nicely because clients can use indirect proxies that contain no endpoint dependencies. The registry resolves indirect proxies using the endpoint information supplied by object adapters each time they are activated.
- `Ice.Default.Locator`
  The server requires a value for this property in order to register its object adapter.

# Starting the Registry 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:

```
$ icegridregistry --Ice.Config=/opt/ripper/registry.cfg
```

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

# Starting the Ripper Server

With the registry up and running, we can now start the server. At a command prompt, we run the program and pass an `--Ice.Config` option indicating the location of the configuration file:

```
$ /opt/ripper/bin/server --Ice.Config=/opt/ripper/server.cfg
```

# Ripper Progress Review

This example demonstrated how to use IceGrid's location service, which is a core component of IceGrid's feature set. By incorporating IceGrid into our application, the client is now able to locate the `MP3EncoderFactory` object using only an indirect proxy and a value for `Ice.Default.Locator`. Furthermore, we can reconfigure the application in any number of ways without modifying the client's code or configuration.

For some applications, the functionality we have already achieved using IceGrid may be entirely sufficient. However, we have only just begun to explore IceGrid's capabilities, and there is much we can still do to improve our application. The next section shows how we can avoid the need to start our server manually by deploying our application onto an IceGrid node.

## See Also

- Locator Configuration for a Client
- Resource Allocation using IceGrid Sessions
- Well-Known Registry Objects
- Using IceGrid Deployment
- Object Adapter Endpoints
- icegridregistry
- IceGrid Properties


Copyright © 2017, ZeroC, Inc.