IceGrid and the Administrative Facility

The Ice administrative facility provides a general purpose solution for administering individual Ice programs. IceGrid extends this functionality in several convenient ways:

- IceGrid automatically enables the facility in deployed servers.
- IceGrid uses the `Process` facet to terminate an active server, giving it an opportunity to perform an orderly shutdown.
- IceGrid provides a secure mechanism for invoking administrative operations on deployed servers.
- IceGrid administrative tools use the facility to display the properties of servers and services, and manipulate and monitor IceBox services.

We discuss each of these items in separate sections below.

On this page:

Enabling the Administrative Facility for a Deployed Server

As we saw in our deployment example, the configuration properties for a deployed server include definitions for the following properties:

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

In conjunction with the `Ice.Default.Locator` property, these definitions satisfy the requirements for enabling the administrative object adapter.

Endpoints

If a server's descriptor does not supply a value for `Ice.Admin.Endpoints`, IceGrid supplies the default value shown below:

```
Ice.Admin.Endpoints=tcp -h 127.0.0.1
```

For security reasons, IceGrid specifies the local host interface (`127.0.0.1`) so that administrative access is limited to clients running on the same host. This configuration permits the IceGrid node to invoke operations on the server's `admin` object, but prevents remote access unless the client establishes an IceGrid administrative session.

Specifying a fixed port is unnecessary because the server registers its endpoints with IceGrid upon each new activation.

Deactivating a Deployed Server

An IceGrid node uses the `Ice::Process` interface to gracefully deactivate a server. In programs using Ice 3.3 or later, this interface is implemented by the administrative facet named `Process`. In earlier versions of Ice, an object adapter implemented this interface in a special servant if the adapter's `RegisterProcess` property was enabled.

Regardless of version, the Ice run time registers an `Ice::Process` proxy with the IceGrid registry when properly configured. Registration normally occurs during communicator initialization, but it can be delayed when a server needs to install its own administrative facets.

When the node is ready to deactivate a server, it invokes the `shutdown` operation on the server's `Ice::Process` proxy. If the server does not terminate in a timely manner, the node asks the operating system to terminate the process. Each server can be configured with its own deactivation timeout. If no timeout is configured, the node uses the value of the property `IceGrid.Node.WaitTime`, which defaults to `60` seconds.

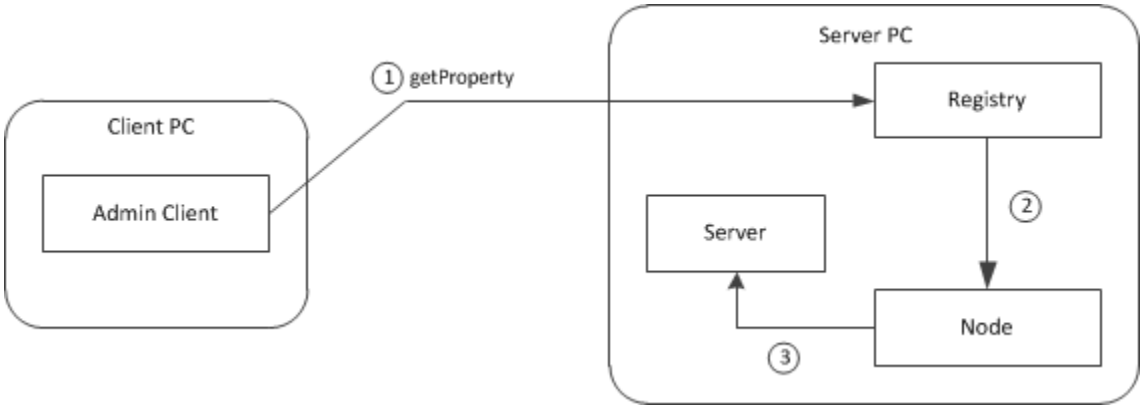
If a server does not register an `Ice::Process` proxy, the IceGrid node cannot request a graceful termination and must resort instead to a more drastic, and potentially harmful, alternative by asking the operating system to terminate the server's process. On Unix, the node sends the `SIGTERM` signal to the process and, if the server does not terminate within the deactivation timeout period, sends the `SIGKILL` signal.

On Windows, the node first sends a `Ctrl+Break` event to the server and, if the server does not stop within the deactivation timeout period, terminates the process immediately.

Servers that disable the `Process` facet can install a signal handler in order to intercept the node's notification about pending deactivation. For example, portable C++ programs could use the `IceUtil::CtrlCHandler` class for this purpose. However, we recommend that servers be allowed to use the `Process` facet when possible.

Routing Administrative Requests

IceGrid defaults to using the local host interface when defining the endpoints of a deployed server's administrative object adapter. This configuration allows local clients such as the IceGrid node to access the server's `admin` object while preventing direct invocations from remote clients. A server's a `dmin` object may still be accessed remotely, but only by clients that establish an IceGrid administrative session. To facilitate these requests, IceGrid uses an intermediary object that relays requests to the server via its node. For example, the following figure illustrates the path of a `getProperty` invocation:



Routing for administrative requests on a server.

Obtaining a Proxy

During an administrative session, a client has two ways of obtaining the intermediary proxy for a server's `admin` object:

Slice

```
module IceGrid {
    interface Admin {
        idempotent string getServerAdminCategory();
        idempotent Object* getServerAdmin(string id)
            throws ServerNotExistException,
                   NodeUnreachableException,
                   DeploymentException;
        // ...
    };
};
```

If the client wishes to construct the proxy itself and already knows the server's ID, the client need only modify the proxy of the `IceGrid::Admin` object with a new identity. The identity's category must be the return value of `getServerAdminCategory`, while its name is the ID of the desired server. The example below demonstrates how to create the proxy and access the `Properties` facet of a server:

C++

```
IceGrid::AdminSessionPrx session = ...;
IceGrid::AdminPrx admin = session->getAdmin();
Ice::Identity serverAdminId;
serverAdminId.category = admin->getServerAdminCategory();
serverAdminId.name = "MyServerId";
Ice::PropertiesAdminPrx props =
    Ice::PropertiesAdminPrx::checkedCast(
        admin->ice_identity(serverAdminId), "Properties");
```

Alternatively, the `getServerAdmin` operation returns a proxy that refers to the `admin` object of the given server. This operation performs additional validation and therefore may raise one of the exceptions shown in its signature above.

Callbacks without Glacier2

IceGrid also supports the relaying of callback requests from a back-end server to an administrative client over the client's existing connection to the registry, which is especially important for a client using a network port that is forwarded by a firewall or protected by a secure tunnel.

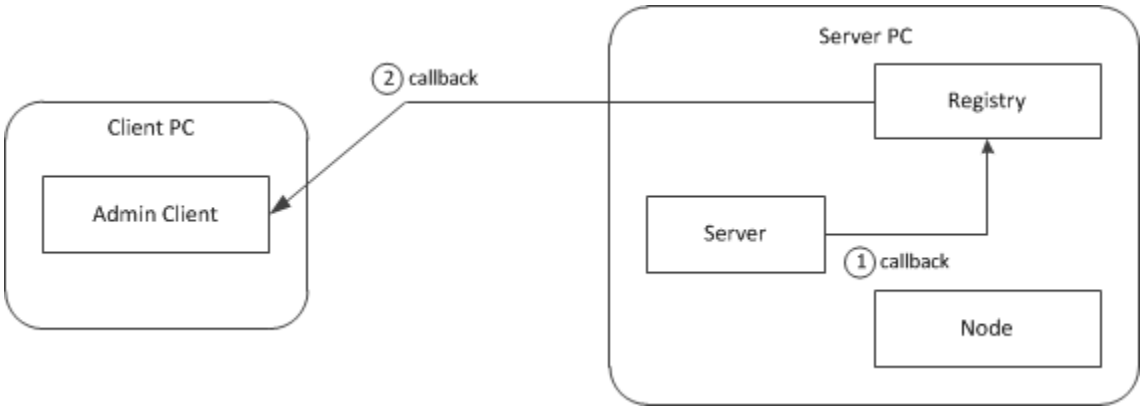
For this mechanism to work properly, a client that established its administrative session directly with IceGrid and not via a Glacier2 router must take additional steps to ensure that the proxies for its callback objects contain the proper identities and endpoints. The `IceGrid::AdminSession` interface provides an operation to help with the client's preparations:

Slice

```
module IceGrid {
interface AdminSession ... {
    idempotent Object* getAdminCallbackTemplate();
    // ...
};
};
```

As its name implies, the `getAdminCallbackTemplate` operation returns a *template proxy* that supplies the identity and endpoints a client needs to configure its callback objects. The information contained in the template proxy is valid for the lifetime of the administrative session. This operation returns a null proxy if the client's administrative session was established via a Glacier2 router, in which case the client should use the callback strategy described in the next section instead.

The endpoints contained in the template proxy are those of an object adapter in the IceGrid registry. The client must transfer these endpoints to the proxies for its callback objects so that callback requests from a server are sent first to IceGrid and then relayed over a bidirectional connection to the client, as shown below:



Routing for callback requests from a server.

Here is the complete list of steps:

1. Invoke `getAdminCallbackTemplate` to obtain the template proxy.
2. Extract the category from the template proxy's identity and use it in all callback objects.
3. Extract the endpoints from the template proxy and use them to establish the published endpoints of the callback object adapter.

4. Create the callback object adapter and associate it with the administrative session's connection, thereby establishing a bidirectional connection with IceGrid.
5. Add servants to the callback object adapter.

As an example, let us assume that we have deployed an IceBox server with the server id `icebox1` and our objective is to register a `ServiceObserver` callback that monitors the state of the IceBox services. The first step is to obtain a proxy for the administrative facet named `IceBox.ServiceManager`:

C++

```
IceGrid::AdminSessionPrx session = ...;
IceGrid::AdminPrx admin = session->getAdmin();
Ice::ObjectPrx obj = admin->getServerAdmin("icebox1");
IceBox::ServiceManagerPrx svcmgr =
    IceBox::ServiceManagerPrx::checkedCast(
        obj, "IceBox.ServiceManager");
```

Next, we retrieve the template proxy and compose the published endpoints for our callback object adapter:

C++

```
Ice::ObjectPrx tmpl = admin->getAdminCallbackTemplate();
Ice::EndpointSeq endpts = tmpl->ice_getEndpoints();
string publishedEndpoints;
for (Ice::EndpointSeq::const_iterator p = endpts.begin(); p != endpts.end(); ++p) {
    if (p == endpts.begin())
        publishedEndpoints = (*p)->toString();
    else
        publishedEndpoints += ":" + (*p)->toString();
}
communicator->getProperties()->setProperty(
    "CallbackAdapter.PublishedEndpoints", publishedEndpoints);
```

The final steps involve creating the callback object adapter, adding a servant, establishing the bidirectional connection and registering our callback with the service manager:

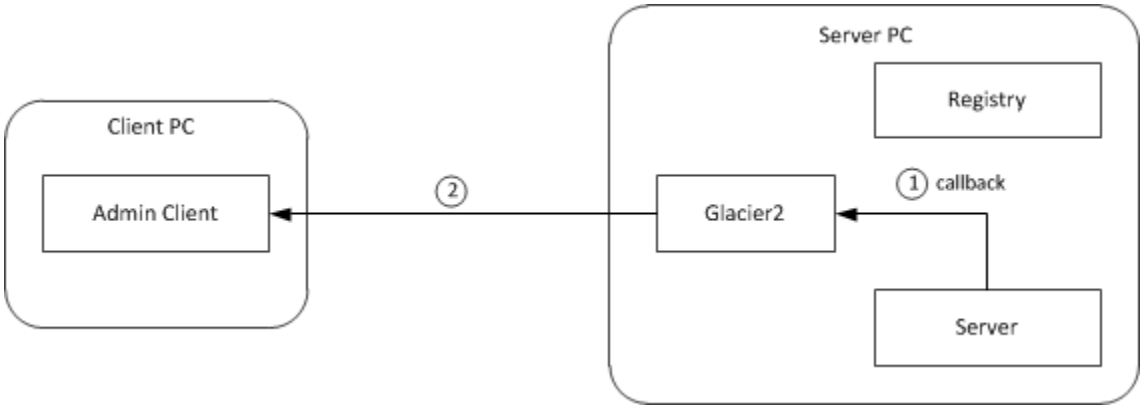
C++

```
Ice::ObjectAdapterPtr callbackAdapter = communicator->createObjectAdapter("CallbackAdapter");
Ice::Identity cbid;
cbid.category = tmpl->ice_getIdentity().category;
cbid.name = "observer";
IceBox::ServiceObserverPtr obs = new ObserverI;
Ice::ObjectPrx cbobj = callbackAdapter->add(obs, cbid);
IceBox::ServiceObserverPrx cb = IceBox::ServiceObserverPrx::uncheckedCast(cbobj);
callbackAdapter->activate();
session->ice_getConnection()->setAdapter(callbackAdapter);
svcmgr->addObserver(cb);
```

At this point the client is ready to receive callbacks from the IceBox server whenever one of its services changes state.

Callbacks with Glacier2

A client that creates an administrative session via a Glacier2 router already has a bidirectional connection over which callbacks from administrative facets are relayed. The flow of requests is shown in the illustration below, which presents a simplified view with the router and IceGrid services all running on the same host.



Routing for callback requests from a server.

To prepare for receiving callbacks, the client must perform the same steps as for any router client:

1. Obtain a proxy for the router.
2. Retrieve the category to be used in callback objects.
3. Create the callback object adapter and associate it with the router, thereby establishing a bidirectional connection.
4. Add servants to the callback object adapter.

Repeating the example from the previous section, we assume that we have deployed an IceBox server with the server ID `icebox1` and our objective is to register a `ServiceObserver` callback that monitors the state of the IceBox services. The first step is to obtain a proxy for the administrative facet named `IceBox.ServiceManager`:

C++

```
IceGrid::AdminSessionPrx session = ...;
IceGrid::AdminPrx admin = session->getAdmin();
Ice::ObjectPrx obj = admin->getServerAdmin("icebox1");
IceBox::ServiceManagerPrx svcmgr =
    IceBox::ServiceManagerPrx::checkedCast(obj, "IceBox.ServiceManager");
```

Now we are ready to create the object adapter and register the observer:

C++

```
Ice::RouterPrx router = communicator->getDefaultRouter();
Ice::ObjectAdapterPtr callbackAdapter =
    communicator->createObjectAdapterWithRouter("CallbackAdapter", router);
Ice::Identity cbid;
cbid.category = router->getCategoryForClient();
cbid.name = "observer";
IceBox::ServiceObserverPtr obs = new ObserverI;
Ice::ObjectPrx cbobj = callbackAdapter->add(obs, cbid);
IceBox::ServiceObserverPrx cb = IceBox::ServiceObserverPrx::uncheckedCast(cbobj);
callbackAdapter->activate();
svcmgr->addObserver(cb);
```

At this point the client is ready to receive callbacks from the IceBox server whenever one of its services changes state.

Using the Administrative Facility in IceGrid Utilities

This section discusses the ways in which the IceGrid utilities make use of the administrative facility.

Properties

The command line and graphical utilities allow you to explore the configuration properties of a server or service.

One property in particular, `BuildId`, is given special consideration by the graphical utility. Although it is not used by the Ice run time, the `BuildId` property gives you the ability to describe the build configuration of your application. The property's value is shown by the graphical utility in its own field in the attributes of a server or service, as well as in the list of properties. You can also retrieve the value of this property using the command-line utility with the following statement:

```
> server property MyServerId BuildId
```

Or, for an IceBox service, with this command:

```
> service property MyServerId MyService BuildId
```

The utilities use the `Properties` facet to access these properties, via a proxy obtained as described above.

Administering IceBox Services

IceBox provides an administrative facet that implements the `IceBox::ServiceManager` interface, which supports operations for stopping an active service, and for starting a service that is currently inactive. These operations are available in both the command line and graphical utilities.

IceBox also defines a `ServiceObserver` interface for receiving callbacks when services are stopped or started. The graphical utility implements this interface so that it can present an updated view of the state of an IceBox server. We presented examples that demonstrate how to register an observer with the IceBox administrative facet.

See Also

- Administrative Facility
- The Process Facet
- The Properties Facet
- The Administrative Object Adapter
- The admin Object
- Custom Administrative Facets
- Security Considerations for Administrative Facets
- Portable Signal Handling in C++
- Bidirectional Connections
- Using IceGrid Deployment
- Glacier2 Integration with IceGrid
- IceGrid Administrative Sessions
- icegridadmin Command Line Tool
- Callbacks through Glacier2
- IceBox
- IceBox Administration
- IceGrid Properties