

The Process Facet

An activation service, such as an [IceGrid](#) node, needs a reliable way to gracefully deactivate a server. One approach is to use a platform-specific mechanism, such as POSIX signals. This works well on POSIX platforms when the server is prepared to [intercept signals](#) and react appropriately. On Windows platforms, it works less reliably for C++ servers, and not at all for Java servers. For these reasons, the `Process` facet provides an alternative that is both portable and reliable.



Be aware of the [security considerations](#) associated with enabling the `Process` facet.

On this page:

- [The Process Interface](#)
- [Application Requirements for the Process Facet](#)
- [Replacing the Process Facet](#)
- [Integrating the Process Facet with an Activation Service](#)

The Process Interface

The Slice interface `Ice::Process` allows an activation service to request a graceful shutdown of the program:

Slice

```
module Ice {
  interface Process {
    void shutdown();
    void writeMessage(string message, int fd);
  };
};
```

When `shutdown` is invoked, the object implementing this interface is expected to initiate the termination of its process. The activation service may expect the program to terminate within a certain period of time, after which it may terminate the program abruptly.

The `writeMessage` operation allows remote clients to print a message to the program's standard output (`fd == 1`) or standard error (`fd == 2`) channels.

Application Requirements for the Process Facet

The default implementation of the `Process` facet requires cooperation from an application in order to successfully terminate a process. Specifically, the facet invokes `shutdown` on its [communicator](#) and assumes that the application uses this event as a signal to commence its termination procedure. For example, an application typically uses a thread (often the main thread) to call the communicator operation `waitForShutdown`, which blocks the calling thread until the communicator is shut down or destroyed. After `waitForShutdown` returns, the calling thread can initiate a graceful shutdown of its process.

Replacing the Process Facet

You can replace the default `Process` facet if your application requires a different scheme for gracefully shutting itself down. To define your own facet, create a servant that implements the `Ice::Process` interface. As an example, the servant definition shown below duplicates the functionality of the default `Process` facet:

C++

```

class ProcessI : public Ice::Process {
public:
    ProcessI(const Ice::CommunicatorPtr& communicator) : _communicator(communicator)
    {}

    void shutdown(const Ice::Current&)
    {
        _communicator->shutdown();
    }

    void writeMessage(const string& msg, Ice::Int fd, const Ice::Current&)
    {
        if(fd == 1) cout << msg << endl;
        else if(fd == 2) cerr << msg << endl;
    }

private:
    const Ice::CommunicatorPtr _communicator;
};

```

As you can see, the default implementation of `shutdown` simply shuts down the communicator, which initiates an orderly termination of the Ice run time's server-side components and prevents object adapters from dispatching any new requests. You can add your own application-specific behavior to the `shutdown` method to ensure that your program terminates in a timely manner.



A servant **must not invoke `destroy`** on its communicator while executing a dispatched operation.

To avoid the risk of a race condition, the recommended strategy for replacing the `Process` facet is to delay creation of the administrative facets so that your application has a chance to replace the facet:

```
Ice.Admin.DelayCreation=1
```

With `Ice.Admin.DelayCreation` enabled, the application can safely remove the default `Process` facet and install its own:

C++

```

Ice::CommunicatorPtr communicator = ...
communicator->removeAdminFacet("Process");
Ice::ProcessPtr myProcessFacet = new MyProcessFacet(...);
communicator->addAdminFacet(myProcessFacet, "Process");

```

The final step is to activate the administrative facility by calling `getAdmin` on the communicator:

C++

```
communicator->getAdmin();
```

Integrating the Process Facet with an Activation Service

If the `Ice.Admin.ServerId` and `Ice.Default.Locator` properties are defined, the Ice run time performs the following steps after creating the `Ice.Admin` object adapter:

- Obtains proxies for the `Process` facet and the default locator
- Invokes `getRegistry` on the proxy to obtain a proxy for the locator registry

- Invokes `setServerProcessProxy` on the locator registry and supplies the value of `Ice.Admin.ServerId` along with a proxy for the `Process facet`

The identifier specified by `Ice.Admin.ServerId` must uniquely identify the process within the locator registry.

In the case of [IceGrid](#), the node defines the `Ice.Admin.ServerId` and `Ice.Default.Locator` properties for each deployed server. The node also supplies a value for `Ice.Admin.Endpoints` if the property is not defined by the server.

See Also

- [Communicators](#)
- [Facets and Versioning](#)
- [Portable Signal Handling in C++](#)
- [Security Considerations for Administrative Facets](#)
- [Object Adapters](#)
- [The Administrative Object Adapter](#)
- [Ice Administrative Properties](#)
- [IceGrid](#)