Communicators

Previous Next

The main entry point to the Ice run time is represented by the local Slice interface `Ice::Communicator`. An instance of `Ice::Communicator` is associated with a number of run-time resources:

- Client-side thread pool
  The client-side thread pool is used to process replies to asynchronous method invocations (AMI), to avoid deadlocks in callbacks, and to process incoming requests on bidirectional connections.

- Server-side thread pool
  Threads in this pool accept incoming connections and handle requests from clients.

- Configuration properties
  Various aspects of the Ice run time can be configured via properties. Each communicator has its own set of such configuration properties.

- Object factories
  In order to instantiate classes that are derived from a known base type, the communicator maintains a set of object factories that can instantiate the class on behalf of the Ice run time. Object factories are discussed in each client-side language mapping.

- Logger object
  A logger object implements the `Ice::Logger` interface and determines how log messages that are produced by the Ice run time are handled.

- Default router
  A router implements the `Ice::Router` interface. Routers are used by Glacier2 to implement the firewall functionality of Ice.

- Default locator
  A locator is an object that resolves an object identity to a proxy. A locator object is implemented by a location service.

- Plug-in manager
  Plug-ins are objects that add features to a communicator. For example, IceSSL is implemented as a plug-in. Each communicator has a plug-in manager that implements the `Ice::PluginManager` interface and provides access to the set of plug-ins for a communicator.

- Object adapters
  Object adapters dispatch incoming requests and take care of passing each request to the correct servant.

Object adapters and objects that use different communicators are completely independent from each other. Specifically:

- Each communicator uses its own thread pool. This means that if, for example, one communicator runs out of threads for incoming requests, only objects using that communicator are affected. Objects using other communicators have their own thread pool and are therefore unaffected.
- Collocated invocations across different communicators are not optimized, whereas collocated invocations using the same communicator bypass much of the overhead of call dispatch.

Typically, servers use only a single communicator but, occasionally, multiple communicators can be useful. For example, IceBox, uses a separate communicator for each Ice service it loads to ensure that different services cannot interfere with each other. Multiple communicators are also useful to avoid thread starvation: if one service runs out of threads, this leaves the remaining services unaffected.

The communicator's interface is defined in Slice. Part of this local interface looks as follows:

Slice

```
module Ice
{
    ["clr:implements:_System.IDisposable", "java:implements:java.lang.AutoCloseable", "php:internal"]
    local interface Communicator
    {
        string proxyToString(Object* obj);
        Object* stringToProxy(string str);
        PropertyDict proxyToProperty(Object* proxy, string property);
        Object* propertyToProxy(string property);
        string identityToString(Identity id);
        ObjectAdapter createObjectAdapter(string name);
        ObjectAdapter createObjectAdapterWithEndpoints(string name, string endpoints);
        ObjectAdapter createObjectAdapterWithRouter(string name, Router* rtr);
        void shutdown();
        void waitForShutdown();
        bool isShutdown();
        ["cpp:noexcept"] void destroy();
        // ...
    }
}
```

The communicator offers a number of operations:

- `proxyToString`
  `stringToProxy`
  These operations allow you to convert a proxy into its stringified representation and vice versa. Instead of calling `proxyToString` on the communicator, you can also use the `ice_toString` proxy method to stringify it. However, you can only stringify non-null proxies that way — to stringify a null proxy, you must use `proxyToString`. (The stringified representation of a null proxy is the empty string.) `proxyToString` and `ice_toString` stringify non-printable ASCII characters and non-ASCII characters in the proxy's identity, facet and object adapter id as specified through the `Ice.ToStringMode` property.

- `proxyToProperty`
  `propertyToProxy`
  The `proxyToProperty` operation returns the set of proxy properties for the supplied proxy. The `property` parameter specifies the base name for the properties in the returned set. `propertyToProxy` retrieves the configuration property with the given name and converts its value into a proxy. A null proxy is returned if no property is found with the specified name.

- `identityToString`
  This operation converts an identity to a string. The `Ice.ToStringMode` property controls how non-printable ASCII characters and non-ASCII characters are represented in the resulting string.

- `createObjectAdapter`
  `createObjectAdapterWithEndpoints`
  `createObjectAdapterWithRouter`
  These operations create a new object adapter. Each object adapter is associated with zero or more transport endpoints. Typically, an object adapter has a single transport endpoint. However, an object adapter can also offer multiple endpoints. If so, these endpoints each lead to the same set of objects and represent alternative means of accessing these objects. This is useful, for example, if a server is behind a firewall but must offer access to its objects to both internal and external clients; by binding the adapter to both the internal and external interfaces, the objects implemented in the server can be accessed via either interface.

  An object adapter also can have no endpoint at all. In that case, the adapter can only be reached via collocated invocations originating from the same communicator as is used by the adapter.

  Whereas `createObjectAdapter` determines its transport endpoints from configuration information, `createObjectAdapterWithEndpoints` allows you to supply the transport endpoints for the new adapter. Typically, you should use `createObjectAdapter` in preference to `createObjectAdapterWithEndpoints`. Doing so keeps transport-specific information, such as host names and port numbers, out of the source code and allows you to reconfigure the application by changing a property (and so avoid recompilation when a transport endpoint needs to be changed).

  `createObjectAdapterWithRouter` creates a routed object adapter that allows clients to receive callbacks from servers that are behind a router.

  The newly-created adapter uses its name as a prefix for a collection of configuration properties that tailor the adapter's behavior. By default, the adapter prints a warning if other properties are defined having the same prefix, but you can disable this warning using the property `Ice.Warn.UnknownProperties`.

- `shutdown`
  This operation shuts down the server side of the Ice run time:
  - Operation invocations that are in progress at the time `shutdown` is called are allowed to complete normally. `shutdown` does *not* wait for these operations to complete; when `shutdown` returns, you know that no new incoming requests will be dispatched, but operations that were already in progress at the time you called `shutdown` may still be running. You can wait for still-executing operations to complete by calling `waitForShutdown`.

    - Operation invocations that arrive after the server has called `shutdown` either fail with a `ConnectFailedException` or are transparently redirected to a new instance of the server (via IceGrid).
    - Note that `shutdown` initiates deactivation of all object adapters associated with the communicator, so attempts to use an adapter once `shutdown` has completed raise an `ObjectAdapterDeactivatedException`.

- `waitForShutdown`
  On the server side, this operation suspends the calling thread until the communicator has shut down (that is, until no more operations are executing in the server). This allows you to wait until the server is idle before you destroy the communicator.
  On the client side, `waitForShutdown` simply waits until another thread has called `shutdown` or `destroy`.

- `isShutdown`
  This operation returns true if `shutdown` has been invoked on the communicator. A return value of true does not necessarily indicate that the shutdown process has completed, only that it has been initiated. An application that needs to know whether shutdown is complete can call `waitForShutdown`. If the blocking nature of `waitForShutdown` is undesirable, the application can invoke it from a separate thread.

- `destroy`
  This operation destroys the communicator and all its associated resources, such as threads, communication endpoints, object adapters, and memory resources. Once you have destroyed the communicator (and therefore destroyed the run time for that communicator), you must not call any other Ice operation (other than to create another communicator).
  It is imperative that you call `destroy` before you leave the `main` function of your program. Failure to do so results in undefined behavior. Calling `destroy` before leaving `main` is necessary because `destroy` waits for all running threads to terminate before it returns. If you leave `main` without calling `destroy`, you will leave `main` with other threads still running; many threading packages do not allow you to do this and end up crashing your program.

  If you call `destroy` without calling `shutdown`, the call waits for all executing operation invocations to complete before it returns (that is, the implementation of `destroy` implicitly calls `shutdown` followed by `waitForShutdown`). `shutdown` (and, therefore, `destroy`) deactivates all object adapters that are associated with the communicator. Since `destroy` blocks until all operation invocations complete, a servant will deadlock if it invokes `destroy` on its own communicator while executing a dispatched operation.

  On the client side, calling `destroy` while operations are still executing causes those operations to terminate with a `CommunicatorDestroyedException`.

  > Some language mappings provide facilities to `destroy` the communicator. For example, in Java the communicator implements `java.lang.AutoCloseable` and in C# it implements the `IDisposable` interface. It enables the code to use Java `try-with-resources` or C# `using` statements to manage the communicator initialization and destruction. Please refer to each language mapping section for more information.

Back to Top ^

See Also

- Properties and Configuration
- Object Adapters
- Object Adapter Endpoints
- Object Identity
- Obtaining Proxies
- Proxy Methods
- Logger Facility
- The Ice Threading Model
- Bidirectional Connections
- Locators
- IceSSL
- Glacier2
- IceBox

Previous

Next