The Server-Side main Function in Objective-C

This page discusses how to initialize and finalize the server-side run time.

On this page:

Initializing and Finalizing the Server-Side Run Time

The main entry point to the Ice run time is represented by the local interface `ICECommunicator`. As for the client side, you must initialize the Ice run time by calling `createCommunicator` (a class method of the `ICEUtil` class) before you can do anything else in your server. `createCommunicator` returns an instance of type `id<ICECommunicator>`:

Objective-C

```
int
main(int argc, char* argv[])
{
    // ...
    id<ICECommunicator> communicator = [ICEUtil createCommunicator:&argc argv:argv];
    // ...
}
```

`createCommunicator` accepts a *pointer* to `argc` as well as `argv`. The class method scans the argument vector for any command-line options that are relevant to the Ice run time; any such options are removed from the argument vector so, when `createCommunicator` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `createCommunicator` throws an exception.

Before leaving your `main` function, you *must* call `Communicator::destroy`. The `destroy` operation is responsible for finalizing the Ice run time. In particular, `destroy` waits for any operation implementations that are still executing in the server to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your `main` function to terminate without calling `destroy` first; doing so has undefined behavior.

The general shape of our server-side `main` function is therefore as follows:

Objective-C

```
#import <Ice/Ice.h>

int
main(int argc, char* argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int status = 1;
    id<ICECommunicator> communicator = nil;
    @try {
        communicator = [ICEUtil createCommunicator:&argc argv:argv];

        // Server code here...

        status = 0;
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    @try {
        [communicator destroy];
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    [pool release];
    return status;
}
```

Note that the code places the call to `createCommunicator` into a `try` block and takes care to return the correct exit status to the operating system. Also note that the code creates and releases an autorelease pool. This ensures that memory will be released before the program terminates.

The `catch` handler for `NSException` ensures that the communicator is destroyed regardless of whether the program terminates normally or due to an exception.

You must *not* release the communicator that is returned by `createCommunicator`. As for any operation that returns a pointer, the Ice run time calls `autorelease` on the returned instance, so you do not have to release it yourself.

This is also the reason why `createCommunicator` is not called `initialize` (as it is for other language mappings) — `initialize` would suggest that the return value must be released because the method name begins with `init`.

Alternative Ways to Create a Communicator in Objective-C

`createCommunicator` is provided in several versions that accept different arguments. Here is the complete list:

- `(id<ICECommunicator>) createCommunicator: (int*)argc argv:(char*[])argv itData:(ICEInitializationData*) initData;`
  This is the designated initializer — the remaining versions of `createCommunicator` are implemented in terms of this initializer. As for the version we saw in the preceding section, this version accepts a pointer to `argc` as well as `argv` and removes Ice-related command-line options from the argument vector. The `initData` argument allows you to pass additional initialization information to the Ice run time (see below).

- `+(id<ICECommunicator>) createCommunicator;`
  This is equivalent to calling:
  `[ICEUtil createCommunicator:nil argv:nil initData:nil];`

- `+(id<ICECommunicator>) createCommunicator: (int*)argc argv:(char*[])argv;`
  This is equivalent to calling
  `[ICEUtil createCommunicator:&argc argv:argv initData:nil];`

- `+(id<ICECommunicator>) createCommunicator: (ICEInitializationData*)initData`
  This is equivalent to calling
  `[ICEUtil createCommunicator:nil argv:nil initData:initData];`

The `initData` argument is of type `ICEInitializationData`. Even though it has no Slice definition, this class behaves as if it were a Slice structure with the following definition:

Slice

```
#include <Properties.ice>
#include <Logger.ice>

["objc:prefix:ICE"]
module Ice {
    dictionary<string, string> PrefixDict;

    local struct InitializationData {
        Ice::Properties properties;
        Ice::Logger logger;
    };
};
```

The `properties` member allows you to explicitly set property values for the communicator to be created. This is useful, for example, if you want to ensure that a particular property setting is always used by the communicator.

The `logger` member sets the logger that the Ice run time uses to log messages. If you do not set a logger (leaving the `logger` member as `nil`), the run time installs a default logger that calls `NSLog` to log messages.

See Also

- Communicator Initialization
- Properties and Configuration
- Logger Facility