Example of a File System Server in Objective-C

This page presents the source code for a C++ server that implements our file system and communicates with the client we wrote earlier. The code here is fully functional, apart from the required interlocking for threads.

The server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

The server code presented here is not quite correct as it stands: if two clients access the same file in parallel, each via a different thread, one thread may read the lines instance variable while another thread updates it. Obviously, if that happens, we may write or return garbage or, worse, crash the server. However, it is trivial to make the read and write operations thread-safe with a few lines of code. We discuss how to write thread-safe servant implementations in The Ice Threading Model.

On this page:

Implementing a File System Server in Objective-C

We have now seen enough of the server-side Objective-C mapping to implement a server for our file system. (You may find it useful to review these Slice definitions before studying the source code.)

Our server is composed of three source files:

- `Server.m`
  This file contains the server main program.
- `FileI.m`
  This file contains the implementation for the `File` servants.
- `DirectoryI.m`
  This file contains the implementation for the `Directory` servants.

Server main Program in Objective-C

Our server main program, in the file `Server.m`, uses the structure we saw in an earlier example:

Objective-C

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

#import <Foundation/NSAutoreleasePool.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];

        id<ICEObjectAdapter> adapter = [communicator createObjectAdapterWithEndpoints:
                                @"SimpleFilesystem"
                                endpoints:@"default -p 10000"];
```

```
        // Create the root directory (with name "/" and no parent)
        //
        DirectoryI *root = [DirectoryI directoryi:@"/" parent:nil];
        [root activate:adapter];

        // Create a file called "README" in the root directory
        //
        FileI *file = [FileI filei:@"README" parent:root];
        NSMutableArray *text = [NSMutableArray arrayWithObject:
            @"This file system contains a collection of poetry."];
        [file write:text current:nil];
        [file activate:adapter];

        // Create a directory called "Coleridge" in the root dir
        //
        DirectoryI *coleridge = [DirectoryI directoryi:@"Coleridge" parent:root];
        [coleridge activate:adapter];

        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = [FileI filei:@"Kubla_Khan" parent:coleridge];
        text = [NSMutableArray arrayWithObjects:
                            @"In Xanadu did Kubla Khan",
                            @"A stately pleasure-dome decree:",
                            @"Where Alph, the sacred river, ran",
                            @"Through caverns measureless to man",
                            @"Down to a sunless sea.",
                            nil];
        [file write:text current:nil];
        [file activate:adapter];

        // All objects are created, allow client requests now
        //
        [adapter activate];

        // Wait until we are done
        //
        [communicator waitForShutdown];

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

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

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

There is quite a bit of code here, so let us examine each section in detail:

Objective-C

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

#import <Foundation/NSAutoreleasePool.h>
```

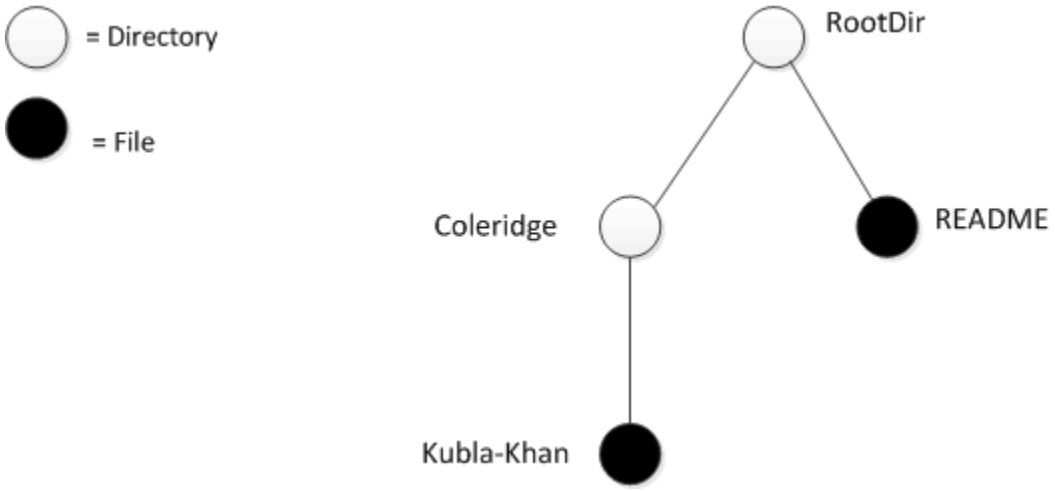
The code includes the header `Ice/Ice.h`, which contains the definitions for the Ice run time, and the files `FileI.h` and `DirectoryI.h`, which contain the definitions of our servant implementations. Because we use an autorelease pool, we need to include `Foundation/NSAutoreleasePool.h` as well.

The next part of the source code is mostly boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`, which blocks the calling thread until you call `shutdown` or `destroy` on the communicator. (Ice does not make any demands on the main thread, so `waitForShutdown` simply blocks the calling thread; if you want to use the main thread for other purposes, you are free to do so.)

Objective-C

```
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];

        id<ICEObjectAdapter> adapter = [communicator createObjectAdapterWithEndpoints:
                                @"SimpleFilesystem"
                                endpoints:@"default -p 10000"];

        // ...

        // All objects are created, allow client requests now
        //
        [adapter activate];

        // Wait until we are done
        //
        [communicator waitForShutdown];

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

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

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

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown below:



A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters: the name of the directory or file to be created and the servant for the parent directory. (For the root directory, which has no parent, we pass a `nil` parent.) Thus, the statement

Objective-C

```
DirectoryI *root = [DirectoryI directoryi:@"/" parent:nil];
```

creates the root directory, with the name `"/"` and no parent directory.

Here is the code that establishes the structure in the above illustration shown:

Objective-C

```
        // Create the root directory (with name "/" and no parent)
        //
        DirectoryI *root = [DirectoryI directoryi:@"/" parent:nil];
        [root activate:adapter];

        // Create a file called "README" in the root directory
        //
        FileI *file = [FileI filei:@"README" parent:root];
        NSMutableArray *text = [NSMutableArray arrayWithObject:
            @"This file system contains a collection of poetry."];
        [file write:text current:nil];
        [file activate:adapter];

        // Create a directory called "Coleridge" in the root dir
        //
        DirectoryI *coleridge = [DirectoryI directoryi:@"Coleridge" parent:root];
        [coleridge activate:adapter];

        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = [FileI filei:@"Kubla_Khan" parent:coleridge];
        text = [NSMutableArray arrayWithObjects:
                            @"In Xanadu did Kubla Khan",
                            @"A stately pleasure-dome decree:",
                            @"Where Alph, the sacred river, ran",
                            @"Through caverns measureless to man",
                            @"Down to a sunless sea.",
                            nil];
        [file write:text current:nil];
        [file activate:adapter];
```

We first create the root directory and a file `README` within the root directory. (Note that we pass the servant for the root directory as the parent pointer when we create the new node of type `FileI`.)

After creating each servant, the code calls `activate` on the servant. (We will see the definition of this member function shortly.) The `activate` member function adds the servant to the ASM.

The next step is to fill the file with text:

Objective-C

```
        // Create the root directory (with name "/" and no parent)
        //
        DirectoryI *root = [DirectoryI directoryi:@"/" parent:nil];
        [root activate:adapter];

        // Create a file called "README" in the root directory
        //
        FileI *file = [FileI filei:@"README" parent:root];
        NSMutableArray *text = [NSMutableArray arrayWithObject:
            @"This file system contains a collection of poetry."];
        [file write:text current:nil];
        [file activate:adapter];

        // Create a directory called "Coleridge" in the root dir
        //
        DirectoryI *coleridge = [DirectoryI directoryi:@"Coleridge" parent:root];
        [coleridge activate:adapter];

        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = [FileI filei:@"Kubla_Khan" parent:coleridge];
        text = [NSMutableArray arrayWithObjects:
                            @"In Xanadu did Kubla Khan",
                            @"A stately pleasure-dome decree:",
                            @"Where Alph, the sacred river, ran",
                            @"Through caverns measureless to man",
                            @"Down to a sunless sea.",
                            nil];
        [file write:text current:nil];
        [file activate:adapter];
```

Recall that Slice sequences map to `NSArray` or `NSMutableArray`, depending on the parameter direction. Here, we instantiate that array and add a line of text to it.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

Objective-C

```
        [file write:text current:nil];
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via the pointer to the servant (of type `FileI`) and *not* via a proxy (of type `id<FilePrx>`), the Ice run time does not know that this call is even taking place — such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary Objective-C function call. The operation implementation in the servant expects a `current` object. In this case, we pass nil (which is fine because the operation implementation does not use it anyway).

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in the above illustration.

Servant Class Definitions in Objective-C

We must provide servants for the concrete interfaces in our Slice specification, that is, we must provide servants for the `File` and `Directory` interfaces in the Objective-C classes `FileI` and `DirectoryI`. This means that our servant classes look as follows:

Objective-C

```
#import <Filesystem.h>

@interface FileI : FSFile <FSFile>
// ...
@end

@interface DirectoryI : FSDirectory <FSDirectory>
// ...
@end
```

Each servant class derives from its skeleton class and adopts its skeleton protocol.

We now can think about how to implement our servants. One thing that is common to all nodes is that they have a name and a parent directory. As we saw earlier, we pass these details to a convenience constructor, which also takes care of calling `autorelease` on the new servant.

In addition, we will use UUIDs as the object identities for files and directories. This relieves us of the need to otherwise come up with a unique identity for each servant (such as path names, which would only complicate our implementation). Because the `list` operation returns proxies to nodes, and because each proxy carries the identity of the servant it denotes, this means that our servants must store their own identity, so we can create proxies to them when clients ask for them.

For `File` servants, we also need to store the contents of the file, leading to the following definition for the `FileI` class:

Objective-C

```
#import <Filesystem.h>

@class DirectoryI;

@interface FileI : FSFile <FSFile>
{
    @private
        NSString *myName;
        DirectoryI *parent;
        ICEIdentity *ident;
        NSArray *lines;
}

@property(nonatomic, retain) NSString *myName;
@property(nonatomic, retain) DirectoryI *parent;
@property(nonatomic, retain) ICEIdentity *ident;
@property(nonatomic, retain) NSArray *lines;

+(id) filei:(NSString *)name parent:(DirectoryI *)parent;
-(void) write:(NSMutableArray *)text current:(ICECurrent *)current;
-(void) activate:(id<ICEObjectAdapter>)a;
@end
```

The instance variables store the name, parent node, identity, and the contents of the file. The `filei` convenience constructor instantiates the servant, remembers the name and parent directory, assigns a new identity, and calls `autorelease`.

Note that the only Slice operation we have defined here is the `write` method. This is necessary because, as we saw previously, the code in `Server.m` calls this method to initialize the files it creates.

For directories, the requirements are similar. They also need to store a name, parent directory, and object identity. Directories are also responsible for keeping track of the child nodes. We can store these nodes in an array of proxies. This leads to the following definition:

Objective-C

```
#import <Filesystem.h>

@interface DirectoryI : FSDirectory <FSDirectory>
{
    @private
        NSString *myName;
        DirectoryI *parent;
        ICEIdentity *ident;
        NSMutableArray *contents;
}
@property(nonatomic, retain) NSString *myName;
@property(nonatomic, retain) DirectoryI *parent;
@property(nonatomic, retain) ICEIdentity *ident;
@property(nonatomic, retain) NSMutableArray *contents;

+(id) directoryi:(NSString *)name parent:(DirectoryI *)parent;
-(void) addChild:(id<FSNodePrx>)child;
-(void) activate:(id<ICEObjectAdapter>)a;
@end
```

Because the code in `Server.m` does not call any Slice operations on directory servants, we have not declared any of the corresponding methods. (We will see the purpose of the `addChild` method shortly.) As for files, the convenience constructor creates the servant, remembers the name and parent, and assigns an object identity, as well as calling `autorelease`.

Servant Implementation in Objective-C

Let us now turn to how to implement each of the methods for our servants.

Implementing `FileI` in Objective-C

The implementation of the `name`, `read`, and `write` operations for files is trivial, returning or updating the corresponding instance variable:

Objective-C

```
-(NSString *) name:(ICECurrent *)current
{
    return myName;
}

-(NSArray *) read:(ICECurrent *)current
{
    return lines;
}

-(void) write:(NSMutableArray *)text current:(ICECurrent *)current
{
    self.lines = text;
}
```

Note that this constitutes the complete implementation of the Slice operations for files.

Here is the convenience constructor:

Objective-C

```
+(id) filei:(NSString *)name parent:(DirectoryI *)parent
{
    FileI *instance = [[[FileI alloc] init] autorelease];
    if(instance == nil)
    {
        return nil;
    }
    instance.myName = name;
    instance.parent = parent;
    instance.ident = [ICEIdentity identity:[ICEUtil generateUUID] category:nil];
    return instance;
}
```

After allocating and autoreleasing the instance, the constructor initializes the instance variables. The only interesting part of this code is how we create the identity for the servant. `generateUUID` is a class method of the `ICEUtil` class that returns a UUID. We assign this UUID to the `name` member of the identity.

We saw earlier that the server calls `activate` after it creates each servant. Here is the implementation of this method:

Objective-C

```
-(void) activate:(id<ICEObjectAdapter>)a
{
    id<FSNodePrx> thisNode = [FSNodePrx uncheckedCast:[a add:self identity:ident]];
    [parent addChild:thisNode];
}
```

This is how our code informs the Ice run time of the existence of a new servant. The call to `add` on the object adapter adds the servant and object identity to the adapter's servant map. In other words, this step creates the link between the object identity (which is embedded in proxies), and the actual Objective-C class instance that provides the behavior for the Slice operations.

`add` returns a proxy to the servant, of type `id<ICEObjectPrx>`. Because the `contents` instance variable of directory servants stores proxies of type `id<FSNodePrx>` (and `addChild` expects a proxy of that type), we down-cast the returned proxy to `id<FSNodePrx>`. In this case, because we know that the servant we just added to the adapter is indeed a servant that implements the operations on the Slice `Node` interface, we can use an `uncheckedCast`.

The call to `addChild` connects the new file to its parent directory.

Finally, we need a `dealloc` function so we do not leak the memory for the servant's instance variables:

Objective-C

```
-(void) dealloc
{
    [myName release];
    [parent release];
    [ident release];
    [lines release];
    [super dealloc];
}
```

Implementing `DirectoryI` in Objective-C

The implementation of the Slice operations for directories is just as simple as for files:

Objective-C

```
-(NSString *) name:(ICECurrent *)current
{
    return myName;
}

-(NSArray *) list:(ICECurrent *)current
{
    return contents;
}
```

Because the `contents` instance variable stores the proxies for child nodes of the directory, the `list` operation simply returns that variable.

The convenience constructor looks much like the one for file servants:

Objective-C

```
+(id) directoryi:(NSString *)name parent:(DirectoryI *)parent
{
    DirectoryI *instance = [[[DirectoryI alloc] init] autorelease];
    if(instance == nil)
    {
        return nil;
    }
    instance.myName = name;
    instance.parent = parent;
    instance.ident = [ICEIdentity
        identity:(parent ? [ICEUtil generateUUID] : @"RootDir")
        category:nil];
    instance.contents = [[NSMutableArray alloc] init];
    return instance;
}
```

The only noteworthy differences are that, for the root directory (which has no parent), the code uses `"RootDir"` as the identity. (As we saw earlier, the client knows that this is the identity of the root directory and uses it to create its proxy.)

The `addChild` method connects our nodes into a hierarchy by updating the `contents` instance variable. That way, each directory knows which nodes are contained in it:

Objective-C

```
-(void) addChild:(id<FSNodePrx>)child
{
     [contents addObject:child];
}
```

Finally, the `activate` and `dealloc` methods are very much like the corresponding methods for files:

Objective-C

```
-(void) activate:(id<ICEObjectAdapter>)a
{
    id<FSNodePrx> thisNode = [FSNodePrx uncheckedCast:[a add:self identity:ident]];
    [parent addChild:thisNode];
}

-(void) dealloc
{
    [myName release];
    [parent release];
    [ident release];
    [contents release];
    [super dealloc];
}
```

See Also

- Slice for a Simple File System
- Objective-C Mapping for Sequences
- Example of a File System Client in Objective-C
- The Server-Side main Function in Objective-C
- The Ice Threading Model