Example of a File System Server in C-Sharp

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 is fully functional, apart from the required interlocking for threads.

The server is 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 shown 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` data member 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. We discuss thread safety in The Ice Threading Model.

On this page:

Implementing a File System Server in C#

We have now seen enough of the server-side 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.cs`
  This file contains the server main program.
- `DirectoryI.cs`
  This file contains the implementation for the `Directory` servants.
- `FileI.cs`
  This file contains the implementation for the `File` servants.

Server Main Program in C#

Our server main program, in the file `Server.cs`, uses the `Ice.Application` class. The `run` method installs a signal handler, creates an object adapter, instantiates a few servants for the directories and files in the file system, and then activates the adapter. This leads to a main program as follows:

C#

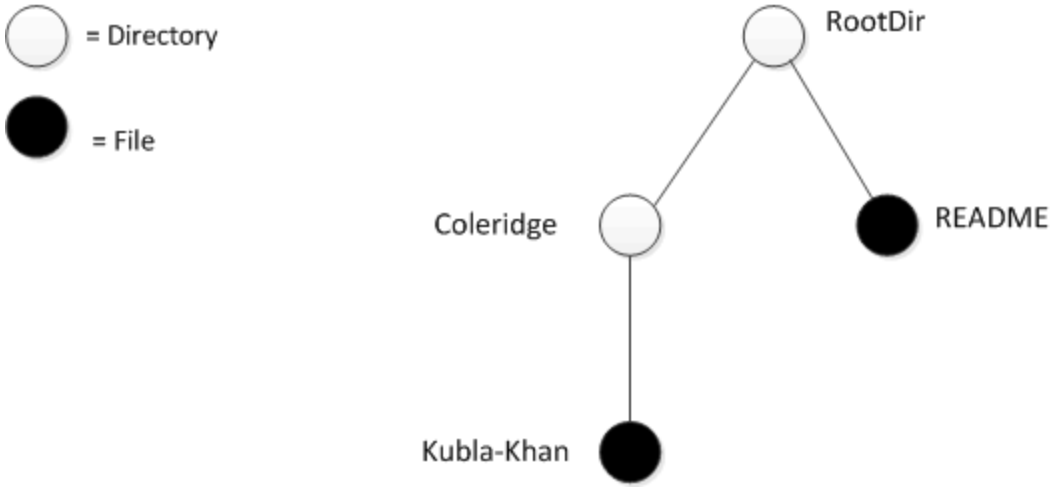
```
using Filesystem;
using System;

public class Server
{
    class App : Ice.Application
    {
        public override int run(string[] args)
        {
            // Terminate cleanly on receipt of a signal
            //
            shutdownOnInterrupt();

            // Create an object adapter (stored in the _adapter
            // static members)
            //
```

```
            Ice.ObjectAdapter adapter = communicator().createObjectAdapterWithEndpoints(
                "SimpleFilesystem", "default -p 10000");
            DirectoryI._adapter = adapter;
            FileI._adapter = adapter;

            // Create the root directory (with name "/" and no
            // parent)
            //
            DirectoryI root = new DirectoryI("/", null);

            // Create a file called "README" in the root directory
            //
            File file = new FileI("README", root);
            string[] text;
            text = new string[] {
                "This file system contains a collection of poetry."
            };
            try {
                file.write(text);
            } catch (GenericError e) {
                Console.Error.WriteLine(e.reason);
            }

            // Create a directory called "Coleridge"
            // in the root directory
            //
            DirectoryI coleridge = new DirectoryI("Coleridge", root);

            // Create a file called "Kubla_Khan"
            // in the Coleridge directory
            //
            file = new FileI("Kubla_Khan", coleridge);
            text = new string[] { "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." };
            try {
                file.write(text);
            } catch (GenericError e) {
                Console.Error.WriteLine(e.reason);
            }

            // All objects are created, allow client requests now
            //
            adapter.activate();

            // Wait until we are done
            //
            communicator().waitForShutdown();

            if (interrupted())
                Console.Error.WriteLine(appName() + ": terminating");

            return 0;
        }
    }

    public static void Main(string[] args)
    {
        App app = new App();
        Environment.Exit(app.main(args));
    }
}
```

The code uses a `using` directive for the `Filesystem` namespace. This avoids having to continuously use fully-qualified identifiers with a `Filesystem.` prefix.

The next part of the source code is the definition of the `Server` class, which includes a nested class that derives from `Ice.Application` and contains the main application logic in its `run` method. Much of this code is boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

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 a reference to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent.) Thus, the statement

C#

```
        DirectoryI root = new DirectoryI("/", null);
```

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

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

C#

```
        // Create the root directory (with name "/" and no parent)
        //
        DirectoryI root = new DirectoryI("/", null);

        // Create a file called "README" in the root directory
        //
        File file = new FileI("README", root);
        string[] text;
        text = new string[] {
            "This file system contains a collection of poetry."
        };
        try {
            file.write(text);
        } catch (GenericError e) {
            Console.Error.WriteLine(e.reason);
        }

        // Create a directory called "Coleridge"
        // in the root directory
        //
        DirectoryI coleridge = new DirectoryI("Coleridge", root);

        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = new FileI("Kubla_Khan", coleridge);
        text = new string[] { "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." };
        try {
            file.write(text);
        } catch (GenericError e) {
            Console.Error.WriteLine(e.reason);
        }
```

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

The next step is to fill the file with text:

C#

```
        string[] text;
        text = new string[] {
            "This file system contains a collection of poetry."
        };
        try {
            file.write(text);
        } catch (GenericError e) {
            Console.Error.WriteLine(e.reason);
        }
```

Recall that Slice sequences by default map to C# arrays. The Slice type `Lines` is simply an array of strings; we add a line of text to our `README` file by initializing the `text` array to contain one element.

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

C#

```
            file.write(text);
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a reference to the servant (of type `FileI`) and not via a proxy (of type `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 C# method call.

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.

`FileI` Servant Class in C#

Our `FileI` servant class has the following basic structure:

C#

```
using Filesystem;
using System;

public class FileI : FileDisp_
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private string _name;
    private DirectoryI _parent;
    private string[] _lines;
}
```

The class has a number of data members:

- `_adapter`
  This static member stores a reference to the single object adapter we use in our server.
- `_name`
  This member stores the name of the file incarnated by the servant.
- `_parent`
  This member stores the reference to the servant for the file's parent directory.
- `_lines`
  This member holds the contents of the file.

The `_name` and `_parent` data members are initialized by the constructor:

C#

```
public FileI(string name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    Debug.Assert(_parent != null);

    // Create an identity
    //
    Ice.Identity myID = new Ice.Identity();
    myId.name = System.Guid.NewGuid().ToString();

    // Add the identity to the object adapter
    //
    _adapter.add(this, myID);

    // Create a proxy for the new node and
    // add it as a child to the parent
    //
    NodePrx thisNode = NodePrxHelper.uncheckedCast(_adapter.createProxy(myID));
    _parent.addChild(thisNode);
}
```

After initializing the `_name` and `_parent` members, the code verifies that the reference to the parent is not null because every file must have a parent directory. The constructor then generates an identity for the file by calling `NewGuid` and adds itself to the servant map by calling `ObjectAdapter.add`. Finally, the constructor creates a proxy for this file and calls the `addChild` method on its parent directory. `addChild` is a helper function that a child directory or file calls to add itself to the list of descendant nodes of its parent directory. We will see the implementation of this function in `DirectoryI` Methods.

The remaining methods of the `FileI` class implement the Slice operations we defined in the `Node` and `File` Slice interfaces:

C#

```
// Slice Node::name() operation

public override string name(Ice.Current current)
{
    return _name;
}

// Slice File::read() operation

public override string[] read(Ice.Current current)
{
    return _lines;
}

// Slice File::write() operation

public override void write(string[] text, Ice.Current current)
{
    _lines = text;
}
```

The `name` method is inherited from the generated `Node` interface (which is a base interface of the `_FileDisp` class from which `FileI` is derived). It simply returns the value of the `_name` member.

The `read` and `write` methods are inherited from the generated `File` interface (which is a base interface of the `_FileDisp` class from which `FileI` is derived) and simply return and set the `_lines` member.

`DirectoryI` Servant Class in C#

The `DirectoryI` class has the following basic structure:

C#

```
using Filesystem;
using System;
using System.Collections;

public class DirectoryI : DirectoryDisp_
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private string _name;
    private DirectoryI _parent;
    private ArrayList _contents = new ArrayList();
}
```

`DirectoryI` Data Members

As for the `FileI` class, we have data members to store the object adapter, the name, and the parent directory. (For the root directory, the `_parent` member holds a null reference.) In addition, we have a `_contents` data member that stores the list of child directories. These data members are initialized by the constructor:

C#

```
    public DirectoryI(string name, DirectoryI parent)
    {
        _name = name;
        _parent = parent;

        // Create an identity. The
        // parent has the fixed identity "RootDir"
        //
        Ice.Identity myID = new Ice.Identity();
        myID.name = _parent != null ? System.Guid.NewGuid().ToString() : "RootDir";

        // Add the identity to the object adapter
        //
        _adapter.add(this, myID);

        // Create a proxy for the new node and
        // add it as a child to the parent
        //
        NodePrx thisNode = NodePrxHelper.uncheckedCast(_adapter.createProxy(myID));
        if (_parent != null)
            _parent.addChild(thisNode);
    }
```

`DirectoryI` Constructor

The constructor creates an identity for the new directory by calling `NewGuid`. (For the root directory, we use the fixed identity `"RootDir"`.) The servant adds itself to the servant map by calling `ObjectAdapter.add` and then creates a proxy to itself and passes it to the `addChild` helper function.

`DirectoryI` Methods

`addChild` simply adds the passed reference to the `_contents` list:

C#

```
public void addChild(NodePrx child)
{
    _contents.Add(child);
}
```

The remainder of the operations, `name` and `list`, are trivial:

C#

```
public override string name(Ice.Current current)
{
    return _name;
}

public override NodePrx[] list(Ice.Current current)
{
    return (NodePrx[])_contents.ToArray(typeof(NodePrx));
}
```

Note that the `_contents` member is of type `System.Collections.ArrayList`, which is convenient for the implementation of the `addChild` method. However, this requires us to convert the list into a C# array in order to return it from the `list` operation.

See Also

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