Example of a File System Server in Python

This page presents the source code for a Python server that implements our file system and communicates with the client we wrote earlier.

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 as 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.

On this page:

Implementing a File System Server in Python

We have now seen enough of the server-side Python 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 implemented in a single source file, `Server.py`, containing our server's main program as well as the definitions of our `Directory` and `File` servant subclasses.

Server Main Program in Python

Our server main program 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:

Python

```python
import sys, threading, Ice, Filesystem

# DirectoryI servant class ...
# FileI servant class ...

class Server(Ice.Application):
    def run(self, args):
        # Terminate cleanly on receipt of a signal
        #
        self.shutdownOnInterrupt()

        # Create an object adapter (stored in the _adapter
        # static members)
        #
        adapter = self.communicator().createObjectAdapterWithEndpoints(
                    "SimpleFilesystem", "default -p 10000")
        DirectoryI._adapter = adapter
        FileI._adapter = adapter

        # Create the root directory (with name "/" and no parent)
        #
        root = DirectoryI("/", None)

        # Create a file called "README" in the root directory
        #
        file = FileI("README", root)
        text = [ "This file system contains a collection of poetry." ]
        try:
            file.write(text)
        except Filesystem.GenericError, e:
            print e.reason

        # Create a directory called "Coleridge"
        # in the root directory
        #
        coleridge = DirectoryI("Coleridge", root)

        # Create a file called "Kubla_Khan"
        # in the Coleridge directory
        #
        file = FileI("Kubla_Khan", coleridge)
        text = [ "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)
        except Filesystem.GenericError, e:
            print e.reason

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

        # Wait until we are done
        #
        self.communicator().waitForShutdown()

        if self.interrupted():
            print self.appName() + ": terminating"

        return 0

app = Server()
sys.exit(app.main(sys.argv))
```

The code defines the `Server` class, which 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:
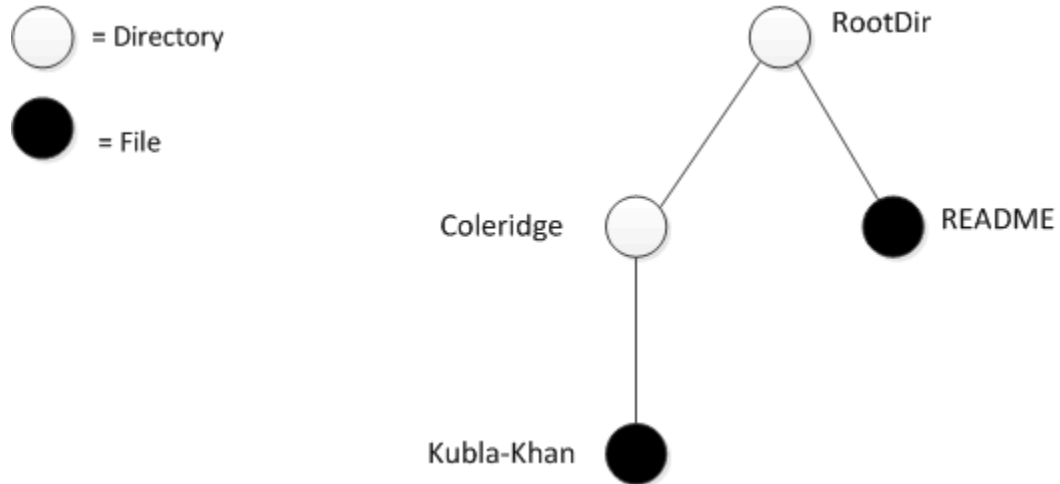
A small file system.

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 `None`.) Thus, the statement

Python

```python
root = DirectoryI("/", None)
```

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

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

Python

```
        # Create the root directory (with name "/" and no parent)
        #
        root = DirectoryI("/", None)

        # Create a file called "README" in the root directory
        #
        file = FileI("README", root)
        text = [ "This file system contains a collection of poetry." ]
        try:
            file.write(text)
        except Filesystem.GenericError, e:
            print e.reason

        # Create a directory called "Coleridge"
        # in the root directory
        #
        coleridge = DirectoryI("Coleridge", root)

        # Create a file called "Kubla_Khan"
        # in the Coleridge directory
        #
        file = FileI("Kubla_Khan", coleridge)
        text = [ "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)
        except Filesystem.GenericError, e:
            print 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:

Python

```
        text = [ "This file system contains a collection of poetry." ]
        try:
            file.write(text)
        except Filesystem.GenericError, e:
            print e.reason
```

Recall that Slice sequences map to Python lists. The Slice type `Lines` is simply a list of strings; we add a line of text to our `README` file by initializing the `text` list to contain one element.

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

Python

```
        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 Python 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 Python

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

Python

```python
class FileI(Filesystem.File):
    # Constructor and operations here...

    _adapter = None
```

The class has a number of data members:

- `_adapter`
  This class member stores a reference to the single object adapter we use in our server.

- `_name`
  This instance member stores the name of the file incarnated by the servant.

- `_parent`
  This instance member stores the reference to the servant for the file's parent directory.

- `_lines`
  This instance member holds the contents of the file.

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

Python

```python
    def __init__(self, name, parent):
        self._name = name
        self._parent = parent
        self._lines = []

        assert(self._parent != None)

        # Create an identity
        #
        myID = Ice.Identity()
        myID.name = Ice.generateUUID()

        # Add ourself to the object adapter
        #
        self._adapter.add(self, myID)

        # Create a proxy for the new node and
        # add it as a child to the parent
        #
        thisNode = Filesystem.NodePrx.uncheckedCast(self._adapter.createProxy(myID))
        self._parent.addChild(thisNode)
```

After initializing the instance members, the code verifies that the reference to the parent is not `None` because every file must have a parent directory. The constructor then generates an identity for the file by calling `Ice.generateUUID` 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:

Python

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

    def name(self, current=None):
        return self._name

    # Slice File::read() operation

    def read(self, current=None):
        return self._lines

    # Slice File::write() operation

    def write(self, text, current=None):
        self._lines = text
```

The `name` method is inherited from the generated `Node` class. It simply returns the value of the `_name` instance member.

The `read` and `write` methods are inherited from the generated `File` class and simply return and set the `_lines` instance member.

`DirectoryI` Servant Class in Python

The `DirectoryI` class has the following basic structure:

Python

```
class DirectoryI(Filesystem.Directory):
    # Constructor and operations here...

    _adapter = None
```

`DirectoryI` Data Members in Python

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 `None`.) In addition, we have a `_contents` data member that stores the list of child directories. These data members are initialized by the constructor:

Python

```python
def __init__(self, name, parent):
    self._name = name
    self._parent = parent
    self._contents = []

    # Create an identity. The
    # parent has the fixed identity "RootDir"
    #
    myID = Ice.Identity()
    if(self._parent):
        myID.name = Ice.generateUUID()
    else:
        myID.name = "RootDir"

    # Add ourself to the object adapter
    #
    self._adapter.add(self, myID)

    # Create a proxy for the new node and
    # add it as a child to the parent
    #
    thisNode = Filesystem.NodePrx.uncheckedCast(self._adapter.createProxy(myID))
    if self._parent:
        self._parent.addChild(thisNode)
```

`DirectoryI` Constructor in Python

The constructor creates an identity for the new directory by calling `Ice.generateUUID`. (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 in Python

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

Python

```python
def addChild(self, child):
    self._contents.append(child)
```

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

Python

```python
def name(self, current=None):
    return self._name

def list(self, current=None):
    return self._contents
```

Thread Safety in Python

The server code we have written so far 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, we can make the `read` and `write` operations thread-safe with a few trivial changes to the `FileI` class:

Python

```python
def __init__(self, name, parent):
    self._name = name
    self._parent = parent
    self._lines = []
    self._mutex = threading.Lock()

    # ...

def name(self, current=None):
    return self._name

def read(self, current=None):
    self._mutex.acquire()
    lines = self._lines[:] # Copy the list
    self._mutex.release()
    return lines

def write(self, text, current=None):
    self._mutex.acquire()
    self._lines = text
    self._mutex.release()
```

We modified the constructor to add the instance member `_mutex`, and then enclosed our `read` and `write` implementations in a critical section. (The `name` method does not require a critical section because the file's name is immutable.)

No changes for thread safety are necessary in the `DirectoryI` class because the `Directory` interface, in its current form, defines no operations that modify the object.

See Also

- Slice for a Simple File System
- Python Mapping for Sequences
- Example of a File System Client in Python
- The Server-Side main Program in Python