CHAPTER 27

Project 8: File Sharing with XML-RPC

This chapter's project is a simple file sharing application. You may be familiar with the concept of file sharing from such applications as the (in)famous Napster (no longer downloadable in its original form), Gnutella (see `http://www.gnutellaforums.com` for discussions about available clients), BitTorrent (available from `http://www.bittorrent.com`), and many others. What you'll be writing is in many ways similar to these, although quite a bit simpler.

The main technology you'll use is XML-RPC. As mentioned in Chapter 15, this is a protocol for calling procedures (functions) remotely, possibly across a network. If you want, you can quite easily use plain socket programming (possibly employing some of the techniques described in Chapters 14 and 24) to implement the functionality of this project. That might even give you better performance, because the XML-RPC protocol does come with a certain overhead. However, XML-RPC is very easy to use and will most likely simplify your code considerably.

What's the Problem?

You want to create a peer-to-peer file sharing program. *File sharing* basically means exchanging files (everything from text files to sound or video clips) between programs running on different machines. *Peer-to-peer* is a term that describes a type of interaction between computer programs that is somewhat different from the common *client-server* interaction (where a client may connect to a server but not vice versa). In a peer-to-peer interaction, any peer may connect to any other. In such a (virtual) network of peers, there is no central authority (as represented by the server in a client/server architecture), which makes the network more robust. It won't collapse unless you shut down most of the peers.

Tip If you're interested in learning more about peer-to-peer systems, try a web search on the phrase "peer-to-peer."

Many issues are involved in constructing a peer-to-peer system. In a system such as the old-school Gnutella, a peer may disseminate a query to all of its neighbors (the other peers it

knows about), and they may subsequently disseminate the query further. Any peer that responds to the query can then send a reply back through the chain of peers to the initial one. The peers work individually and in parallel. More recent systems, such as BitTorrent, use even more clever techniques, such as requiring that you upload files in order to be allowed to download files. To simplify things, this project's system will contact each neighbor in turn, waiting for its response before moving on. This is not quite as efficient as the parallel approach of Gnutella, but good enough for your purposes.

Also, most peer-to-peer systems have clever ways of organizing their structure—that is, which peers are "next to" which—and how this structure evolves over time, as peers connect and disconnect. We'll keep that very simple in this project, but leave things open for improvements.

The following are the requirements that the file sharing program must satisfy:

- Each node must keep track of a set of known nodes, from which it can ask for help. It must be possible for a node to introduce itself to another node (and thereby be included in this set).
- It must be possible to ask a node for a file (by supplying a file name). If the node has the file in question, it should return it; otherwise, it should ask each of its neighbors in turn for the same file (and they, in turn, may ask *their* neighbors). If one of these nodes has the file, it is returned.
- To avoid loops (A asking B, which in turn asks A) and to avoid overly long chains of neighbors asking neighbors (A asking B asking C . . . asking Z), it must be possible to supply a *history* when querying a node. This history is just a list of which nodes have participated in the query up until this point. By not asking nodes already in the history, you avoid loops, and by limiting the length of the history, you avoid overly long query chains.
- There must be some way of connecting to a node and identifying yourself as a trusted party. By doing so, you should be given access to functionality that is not available to untrusted parties (such as other nodes in the peer-to-peer network). This functionality may include asking the node to download and store a file from the other peers in the network (through a query).
- You must have some user interface that lets you connect to a node (as a trusted party) and make it download files. It should be easy to extend and, for that matter, replace this interface.

All of this may seem a bit steep, but as you'll see, implementing it isn't all that hard. And you'll probably find that once you have this in place, adding functionality won't be all that difficult either.

Useful Tools

In this project, you'll use quite a few standard library modules.

The main modules you'll be using are `xmlrpclib` and its close friend `SimpleXMLRPCServer`. The use of `xmlrpclib` is quite straightforward. You simply create a `ServerProxy` object with a URL to the

server, and you immediately have access to the remote procedures. Using `SimpleXMLRPCServer` is a tad more involved, as you'll learn as you work through the project in this chapter.

For the interface to the file sharing program, you'll be using a module from the standard library called `cmd`. To get some (very limited) parallelism, you'll use the `threading` module, and to extract the components of a URL, you'll use the `urlparse` module. These modules are explained later in the chapter.

Other modules you might want to brush up on are `random`, `string`, `time`, and `os.path`. See Chapter 10, as well as the Python Library Reference, for additional details.

Preparations

The libraries used in this project don't require much preparation. If you have a fairly recent version of Python, all of the necessary libraries should be available out of the box.

You don't strictly *have* to be connected to a network to use the software in this project, but it will make things more interesting. If you have access to two (or more) separate machines that are connected (for example, both connected to the Internet), you can run the software on each of these machines and have them communicate with each other (although you may need to make changes to any firewall rules you're running). For testing purposes, it is also possible to run multiple file sharing nodes on the same machine.

First Implementation

Before you can write a first prototype of the `Node` class (a single node or peer in the system), you need to know a bit about how the `SimpleXMLRPCServer` class works. It is instantiated with a tuple of the form `(servername, port)`. The server name is the name of the machine on which the server will run. (You can use an empty string here to indicate localhost, the machine where you're actually executing the program.) The port number can be any port you have access to, typically 1024 and above.

After you have instantiated the server, you may register an instance that implements its "remote methods," with the `register_instance` method. Alternatively, you can register individual functions with the `register_function` method. When you're ready to run the server (so that it can respond to requests from outside), you call its method `serve_forever`. You can easily try this out. Start two interactive Python interpreters. In the first one, enter the following code:

```
>>> from SimpleXMLRPCServer import SimpleXMLRPCServer
>>> s = SimpleXMLRPCServer(("", 4242)) # Localhost at port 4242
>>> def twice(x): # Example function
...     return x*2
...
>>> s.register_function(twice) # Add functionality to the server
>>> s.serve_forever() # Start the server
```

After executing the last statement, the interpreter should seem to "hang." Actually, it's waiting for RPC requests.

To make such a request, switch to the other interpreter and execute the following:

```
>>> from xmlrpclib import ServerProxy # ... or simply Server, if you prefer
>>> s = ServerProxy('http://localhost:4242') # Localhost again...
>>> s.twice(2)
4
```

Pretty impressive, eh? Especially considering that the client part (using `xmlrpclib`) could be run on a different machine. (In that case, you would need to use the actual name of the server machine instead of simply localhost.) As you can see, to access the remote procedures implemented by the server, all that is required is to instantiate a `ServerProxy` with the correct URL. It really couldn't be much easier.

Implementing a Simple Node

Now that we've covered the XML-RPC technicalities, it's time to get started with the coding. (The full source code of the first prototype is found in Listing 27-1, at the end of this section.)

To find out where to begin, it might be a good idea to review your requirements from earlier in this chapter. You're mainly interested in two things: what information must your `Node` hold (attributes) and what actions must it be able to perform (methods)?

The `Node` must have at least the following attributes:

- A directory name, so it knows where to find/store its files.
- A "secret" (or password) that can be used by others to identify themselves (as trusted parties).
- A set of known peers (URLs).
- A URL, which may be added to the query history or possibly supplied to other `Nodes`. (This project won't implement the latter.)

The `Node` constructor will simply set these four attributes. In addition, you'll need a method for querying the `Node`, a method for making it fetch and store a file, and a method to introduce another `Node` to it. Let's call these methods `query`, `fetch`, and `hello`. The following is a sketch of the class, written as pseudocode:

```
class Node:

    def __init__(self, url, dirname, secret):
        self.url = url
        self.dirname = dirname
        self.secret = secret
        self.known = set()

    def query(self, query):
        Look for a file (possibly asking neighbors), and return it as
        a string
```

```
    def fetch(self, query, secret):
        If the secret is correct, perform a regular query and store
        the file. In other words, make the Node find the file and download it.

    def hello(self, other):
        Add the other Node to the known peers
```

Assuming that the set of known URLs is called `known`, the `hello` method is very simple. It just adds `other` to `self.known`, where `other` is the only parameter (a URL). However, XML-RPC requires all methods to return a value; `None` is not accepted. So, let's define two result "codes" that indicate success or failure:

```
OK = 1
FAIL = 2
```

Then the `hello` method can be implemented as follows:

```
    def hello(self, other):
        self.known.add(other)
        return OK
```

When the `Node` is registered with a `SimpleXMLRPCServer`, it will be possible to call this method from the "outside."

The `query` and `fetch` methods are a bit more tricky. Let's begin with `fetch` because it's the simpler of the two. It must take two parameters: the query and the "secret," which is required so that your `Node` can't be arbitrarily manipulated by anyone. Note that calling `fetch` causes the `Node` to download a file. Access to this method should therefore be more restricted than, for example, `query`, which simply passes the file through.

If the supplied secret is not equal to `self.secret` (the one supplied at startup), `fetch` simply returns `FAIL`. Otherwise, it calls `query` to get the file corresponding to the given query (a file name). But what does `query` return? When you call `query`, you would like to know whether the query succeeded, and you would like to have the contents of the relevant file returned if it did. So, let's define the return value of `query` as the pair (tuple) `code, data`, where `code` is either `OK` or `FAIL`, and `data` is the sought-after file (if `code` equals `OK`) stored in a string, or an arbitrary value (for example, an empty string) otherwise.

In `fetch`, the code and the data are retrieved. If the code is `FAIL`, then `fetch` simply returns `FAIL` as well. Otherwise, it opens a new file (in write mode) whose name is the same as the query and which is found in the directory `self.dirname` (you use `os.path.join` to join the two). The data is written to the file, the file is closed, and `OK` is returned. See Listing 27-1 later in this section for the relatively straightforward implementation.

Now, turn your attention to `query`. It receives a query as a parameter, but it should also accept a history (which contains URLs that should not be queried because they are already waiting for a response to the same query). Because this history is empty in the first call to `query`, you can use an empty list as a default value.

If you take a look at the code in Listing 27-1, you'll see that it abstracts away part of the behavior of `query` by creating two utility methods called `_handle` and `_broadcast`. Note that their names begin with underscores, which means that they won't be accessible through

XML-RPC. (This is part of the behavior of `SimpleXMLRPCServer`, not a part of XML-RPC itself.) That is useful because these methods aren't meant to provide separate functionality to an outside party, but are there to structure the code.

For now, let's just assume that `_handle` takes care of the internal handling of a query (checks whether the file exists at this specific `Node`, fetches the data, and so forth) and that it returns a code and some data, just as `query` itself is supposed to. As you can see from the listing, if `code == OK`, then `code, data` is returned immediately—the file was found. However, what should `query` do if the `code` returned from `_handle` is `FAIL`? Then it must ask all other known `Nodes` for help. The first step in this process is to add `self.url` to `history`.

■ **Note** Neither the `+=` operator nor the `append` list method has been used when updating the history because both of these modify lists in place, and you don't want to modify the default value itself.

If the new `history` is too long, `query` returns `FAIL` (along with an empty string). The maximum length is arbitrarily set to 6 and kept in the global constant `MAX_HISTORY_LENGTH`.

WHY IS MAX_HISTORY_LENGTH SET TO 6?

The idea is that any peer in the network should be able to reach another in, at most, six steps. This, of course, depends on the structure of the network (which peers know which), but is supported by the hypothesis of "six degrees of separation," which applies to people and who they know. For a description of this hypothesis, see, for example, Wikipedia's article on six degrees of separation (`http://en.wikipedia.org/wiki/Six_degrees_of_separation`).

Using this number in your program may not be very scientific, but at least it seems like a good guess. On the other hand, in a large network with many nodes, the sequential nature of your program may lead to bad performance for large values of `MAX_HISTORY_LENGTH`, so you might want to reduce it if things get slow.

If `history` isn't too long, the next step is to broadcast the query to all known peers, which is done with the `_broadcast` method. The `_broadcast` method isn't very complicated (see Listing 27-1). It iterates over a copy of `self.known`. If a `peer` is found in `history`, the loop continues to the next peer (using the `continue` statement). Otherwise, a `ServerProxy` is constructed, and the `query` method is called on it. If the query succeeds, its return value is used as the return value from `_broadcast`. Exceptions may occur, due to network problems, a faulty URL, or the fact that the peer doesn't support the `query` method. If such an exception occurs, the peer's URL is removed from `self.known` (in the `except` clause of the `try` statement enclosing the query). Finally, if control reaches the end of the function (nothing has been returned yet), `FAIL` is returned, along with an empty string.

■ **Note** You shouldn't simply iterate over `self.known` because the set may be modified during the iteration. Using a copy is safer.

The `_start` method creates a `SimpleXMLRPCServer` (using the little utility function `getPort`, which extracts the port number from a URL), with `logRequests` set to false (you don't want to keep a log). It then registers `self` with `register_instance` and calls the server's `serve_forever` method.

Finally, the `main` method of the module extracts a URL, a directory, and a secret (password) from the command line; creates a `Node`; and calls its `_start` method.

For the full code of the prototype, see Listing 27-1.

Listing 27-1. *A Simple Node Implementation (simple_node.py)*

```
from xmlrpclib import ServerProxy
from os.path import join, isfile
from SimpleXMLRPCServer import SimpleXMLRPCServer
from urlparse import urlparse
import sys

MAX_HISTORY_LENGTH = 6

OK = 1
FAIL = 2
EMPTY = ''

def getPort(url):
    'Extracts the port from a URL'
    name = urlparse(url)[1]
    parts = name.split(':')
    return int(parts[-1])


class Node:
    """
    A node in a peer-to-peer network.
    """
    def __init__(self, url, dirname, secret):
        self.url = url
        self.dirname = dirname
        self.secret = secret
        self.known = set()
```

```
    def query(self, query, history=[]):
        """
        Performs a query for a file, possibly asking other known Nodes for
        help. Returns the file as a string.
        """
        code, data = self._handle(query)
        if code == OK:
            return code, data
        else:
            history = history + [self.url]
            if len(history) >= MAX_HISTORY_LENGTH:
                return FAIL, EMPTY
            return self._broadcast(query, history)

    def hello(self, other):
        """
        Used to introduce the Node to other Nodes.
        """
        self.known.add(other)
        return OK

    def fetch(self, query, secret):
        """
        Used to make the Node find a file and download it.
        """
        if secret != self.secret: return FAIL
        code, data = self.query(query)
        if code == OK:
            f = open(join(self.dirname, query), 'w')
            f.write(data)
            f.close()
            return OK
        else:
            return FAIL


    def _start(self):
        """
        Used internally to start the XML-RPC server.
        """
        s = SimpleXMLRPCServer(("", getPort(self.url)), logRequests=False)
        s.register_instance(self)
        s.serve_forever()
```

```
    def _handle(self, query):
        """
        Used internally to handle queries.
        """
        dir = self.dirname
        name = join(dir, query)
        if not isfile(name): return FAIL, EMPTY
        return OK, open(name).read()

    def _broadcast(self, query, history):
        """
        Used internally to broadcast a query to all known Nodes.
        """
        for other in self.known.copy():
            if other in history: continue
            try:
                s = ServerProxy(other)
                code, data = s.query(query, history)
                if code == OK:
                    return code, data
            except:
                self.known.remove(other)
        return FAIL, EMPTY

def main():
    url, directory, secret = sys.argv[1:]
    n = Node(url, directory, secret)
    n._start()

if __name__ == '__main__': main()
```

Now let's take a look at a simple example of how this program may be used.

Trying Out the First Implementation

Make sure you have several terminals (xterm, DOS window, or equivalent) open. Let's say you want to run two peers (both on the same machine). Create a directory for each of them, such as `files1` and `files2`. Put a file (for example, `test.txt`) into the `files2` directory. Then, in one terminal, run the following command:

```
python simple_node.py http://localhost:4242 files1 secret1
```

In a real application, you would use the full machine name instead of `localhost`, and you would probably use a secret that is a bit more cryptic than `secret1`.

This is your first peer. Now create another one. In a different terminal, run the following command:

```
python simple_node.py http://localhost:4243 files2 secret2
```

As you can see, this peer serves files from a different directory, uses another port number (4243), and has another secret. If you have followed these instructions, you should have two peers running (each in a separate terminal window). Let's start up an interactive Python interpreter and try to connect to one of them:

```
>>> from xmlrpclib import *
>>> mypeer = ServerProxy('http://localhost:4242') # The first peer
>>> code, data = mypeer.query('test.txt')
>>> code
2
```

As you can see, the first peer fails when asked for the file `test.txt`. (The return code `2` represents failure, remember?) Let's try the same thing with the second peer:

```
>>> otherpeer = ServerProxy('http://localhost:4243') # The second peer
>>> code, data = otherpeer.query('test.txt')
>>> code
1
```

This time, the query succeeds because the file `test.txt` is found in the second peer's file directory. If your test file doesn't contain too much text, you can display the contents of the `data` variable to make sure that the contents of the file have been transferred properly:

```
>>> data
'This is a test\n'
```

So far, so good. How about introducing the first peer to the second one?

```
>>> mypeer.hello('http://localhost:4243') # Introducing mypeer to otherpeer
```

Now the first peer knows the URL of the second, and thus may ask it for help. Let's try querying the first peer again. This time, the query should succeed:

```
>>> mypeer.query('test.txt')
[1, 'This is a test\n']
```

Bingo!

Now there is only one thing left to test: can you make the first node actually download and store the file from the second one?

```
>>> mypeer.fetch('test.txt', 'secret1')
1
```

Well, the return value (`1`) indicates success. And if you look in the `files1` directory, you should see that the file `test.txt` has miraculously appeared. Cool, eh? Feel free to start several peers (on different machines, if you want to), and introduce them to each other. When you grow tired of playing, proceed to the next implementation.

Second Implementation

The first implementation has plenty of flaws and shortcomings. I won't address all of them (some possible improvements are discussed in the section "Further Exploration," at the end of this chapter), but here are some of the more important ones:

- If you try to stop a `Node` and then restart it, you will probably get some error message about the port being in use already.
- You should have a more user-friendly interface than `xmlrpclib` in an interactive Python interpreter.
- The return codes are inconvenient. A more natural and Pythonic solution would be to use a custom exception if the file can't be found.
- The `Node` doesn't check whether the file it returns is actually inside the file directory. By using paths such as `'../somesecretfile.txt'`, a sneaky cracker may get unlawful access to any of your other files.

The first problem is easy to solve. You simply set the `allow_reuse_address` attribute of the `SimpleXMLRPCServer` to true:

```
SimpleXMLRPCServer.allow_reuse_address = 1
```

If you don't want to modify this class directly, you can create your own subclass. The other changes are a bit more involved, and are discussed in the following sections. The source code is shown in Listings 27-2 and 27-3 later in this chapter. (You might want to take a quick look at these listings before reading on.)

Creating the Client Interface

The client interface uses the `Cmd` class from the `cmd` module. For details about how this works, see the Python Library Reference. Simply put, you subclass `Cmd` to create a command-line interface, and implement a method called `do_foo` for each command `foo` you want it to be able to handle. This method will receive the rest of the command line as its only argument (as a string). For example, if you type this in the command-line interface:

```
say hello
```

the method `do_say` is called with the string `'hello'` as its only argument. The prompt of the `Cmd` subclass is determined by the `prompt` attribute.

The only commands implemented in your interface will be `fetch` (to download a file) and `exit` (to exit the program). The `fetch` command simply calls the `fetch` method of the server, printing an error message if the file could not be found. The `exit` commands `print`s an empty line (for aesthetic reasons only) and calls `sys.exit`. (The `EOF` command corresponds to "end of file," which occurs when the user presses Ctrl+D in UNIX.)

But what is all the stuff going on in the constructor? Well, you want each client to be associated with a peer of its own. You *could* simply create a `Node` object and call its `_start` method, but then your `Client` couldn't do anything until the `_start` method returned, which makes the `Client` completely useless. To fix this, the `Node` is started in a separate *thread*. Normally, using threads involves a lot of safeguarding and synchronization with locks and the like. However, because a `Client` interacts with its `Node` only through XML-RPC, you don't need any of this. To run the `_start` method in a separate thread, you just need to put the following code into your program at some suitable place:

```
from threading import Thread
n = Node(url, dirname, self.secret)
t = Thread(target=n._start)
t.start()
```

■ **Caution** You should be careful when rewriting the code of this project. The minute your `Client` starts interacting directly with the `Node` object or vice versa, you may easily run into trouble, because of the threading. Make sure you fully understand threading before you do this.

To make sure that the server is fully started before you start connecting to it with XML-RPC, you'll give it a head start, and wait for a moment with `time.sleep`.

Afterward, you'll go through all the lines in a file of URLs and introduce your server to them with the `hello` method.

You don't really want to be bothered with coming up with a clever secret password. Instead, you can use the utility function `randomString` (in Listing 27-3, shown later in this chapter), which generates a random secret string that is shared between the `Client` and the `Node`.

Raising Exceptions

Instead of returning a code indicating success or failure, you'll just assume success and raise an exception in the case of failure. In XML-RPC, exceptions (or *faults*) are identified by numbers. For this project, I have (arbitrarily) chosen the numbers 100 and 200 for ordinary failure (an unhandled request) and a request refusal (access denied), respectively.

```
UNHANDLED     = 100
ACCESS_DENIED = 200

class UnhandledQuery(Fault):
    """
    An exception that represents an unhandled query.
    """
    def __init__(self, message="Couldn't handle the query"):
        Fault.__init__(self, UNHANDLED, message)
```

```
class AccessDenied(Fault):
    """
    An exception that is raised if a user tries to access a
    resource for which he or she is not authorized.
    """
    def __init__(self, message="Access denied"):
        Fault.__init__(self, ACCESS_DENIED, message)
```

The exceptions are subclasses of `xmlrpclib.Fault`. When they are raised in the server, they are passed on to the client with the same `faultCode`. If an ordinary exception (such as `IOException`) is raised in the server, an instance of the `Fault` class is still created, so you can't simply use arbitrary exceptions here. (Make sure you have a recent version of `SimpleXMLRPCServer`, so it handles exceptions properly.)

As you can see from the source code, the logic is still basically the same, but instead of using `if` statements for checking returned codes, the program now uses exceptions. (Because you can use only `Fault` objects, you need to check the `faultCodes`. If you weren't using XML-RPC, you would have used different exception classes instead, of course.)

Validating File Names

The last issue to deal with is to check whether a given file name is found within a given directory. There are several ways to do this, but to keep things platform-independent (so it works in Windows, in UNIX, and in Mac OS, for example), you should use the module `os.path`.

The simple approach taken here is to create an absolute path from the directory name and the file name (so that, for example, `'/foo/bar/../baz'` is converted to `'/foo/baz'`), the directory name is joined with an empty file name (using `os.path.join`) to ensure that it ends with a file separator (such as `'/'`), and then you check that the absolute file name begins with the absolute directory name. If it does, the file is actually inside the directory.

The full source code for the second implementation is shown Listings 27-2 and 27-3.

Listing 27-2. *A New Node Implementation (server.py)*

```
from xmlrpclib import ServerProxy, Fault
from os.path import join, abspath, isfile
from SimpleXMLRPCServer import SimpleXMLRPCServer
from urlparse import urlparse
import sys

SimpleXMLRPCServer.allow_reuse_address = 1

MAX_HISTORY_LENGTH = 6

UNHANDLED     = 100
ACCESS_DENIED = 200
```

```
class UnhandledQuery(Fault):
    """
    An exception that represents an unhandled query.
    """
    def __init__(self, message="Couldn't handle the query"):
        Fault.__init__(self, UNHANDLED, message)


class AccessDenied(Fault):
    """
    An exception that is raised if a user tries to access a
    resource for which he or she is not authorized.
    """
    def __init__(self, message="Access denied"):
        Fault.__init__(self, ACCESS_DENIED, message)

def inside(dir, name):
    """
    Checks whether a given file name lies within a given directory.
    """
    dir = abspath(dir)
    name = abspath(name)
    return name.startswith(join(dir, ''))

def getPort(url):
    """
    Extracts the port number from a URL.
    """
    name = urlparse(url)[1]
    parts = name.split(':')
    return int(parts[-1])

class Node:
    """
    A node in a peer-to-peer network.
    """
    def __init__(self, url, dirname, secret):
        self.url = url
        self.dirname = dirname
        self.secret = secret
        self.known = set()

    def query(self, query, history=[]):
        """
        Performs a query for a file, possibly asking other known Nodes for
        help. Returns the file as a string.
```

```
    """
    try:
        return self._handle(query)
    except UnhandledQuery:
        history = history + [self.url]
        if len(history) >= MAX_HISTORY_LENGTH: raise
        return self._broadcast(query, history)


def hello(self, other):
    """
    Used to introduce the Node to other Nodes.
    """
    self.known.add(other)
    return 0

def fetch(self, query, secret):
    """
    Used to make the Node find a file and download it.
    """
    if secret != self.secret: raise AccessDenied
    result = self.query(query)
    f = open(join(self.dirname, query), 'w')
    f.write(result)
    f.close()
    return 0

def _start(self):
    """
    Used internally to start the XML-RPC server.
    """
    s = SimpleXMLRPCServer(("", getPort(self.url)), logRequests=False)
    s.register_instance(self)
    s.serve_forever()

def _handle(self, query):
    """
    Used internally to handle queries.
    """
    dir = self.dirname
    name = join(dir, query)
    if not isfile(name): raise UnhandledQuery
    if not inside(dir, name): raise AccessDenied
    return open(name).read()
```

```
    def _broadcast(self, query, history):
        """
        Used internally to broadcast a query to all known Nodes.
        """
        for other in self.known.copy():
            if other in history: continue
            try:
                s = ServerProxy(other)
                return s.query(query, history)

            except Fault, f:
                if f.faultCode == UNHANDLED: pass
                else: self.known.remove(other)
            except:
                self.known.remove(other)
        raise UnhandledQuery

def main():
    url, directory, secret = sys.argv[1:]
    n = Node(url, directory, secret)
    n._start()

if __name__ == '__main__': main()
```

Listing 27-3. *A Node Controller Interface (client.py)*

```
from xmlrpclib import ServerProxy, Fault
from cmd import Cmd
from random import choice
from string import lowercase
from server import Node, UNHANDLED
from threading import Thread
from time import sleep
import sys

HEAD_START = 0.1 # Seconds
SECRET_LENGTH = 100

def randomString(length):
    """
    Returns a random string of letters with the given length.
    """
    chars = []
    letters = lowercase[:26]
```

```
    while length > 0:
        length -= 1
        chars.append(choice(letters))
    return ''.join(chars)

class Client(Cmd):
    """
    A simple text-based interface to the Node class.
    """

    prompt = '> '


    def __init__(self, url, dirname, urlfile):
        """
        Sets the url, dirname, and urlfile, and starts the Node
        Server in a separate thread.
        """
        Cmd.__init__(self)
        self.secret = randomString(SECRET_LENGTH)
        n = Node(url, dirname, self.secret)
        t = Thread(target=n._start)
        t.setDaemon(1)
        t.start()
        # Give the server a head start:
        sleep(HEAD_START)
        self.server = ServerProxy(url)
        for line in open(urlfile):
            line = line.strip()
            self.server.hello(line)

    def do_fetch(self, arg):
        "Call the fetch method of the Server."
        try:
            self.server.fetch(arg, self.secret)
        except Fault, f:
            if f.faultCode != UNHANDLED: raise
            print "Couldn't find the file", arg

    def do_exit(self, arg):
        "Exit the program."
        print
        sys.exit()

    do_EOF = do_exit # End-Of-File is synonymous with 'exit'
```

```
def main():
    urlfile, directory, url = sys.argv[1:]
    client = Client(url, directory, urlfile)
    client.cmdloop()

if __name__ == '__main__': main()
```

Trying Out the Second Implementation

Let's see how the program is used. Start it like this:

```
python client.py urls.txt directory http://servername.com:4242
```

The file `urls.txt` should contain one URL per line—the URLs of all the other peers you know. The directory given as the second argument should contain the files you want to share (and will be the location where new files are downloaded). The last argument is the URL to the peer. When you run this command, you should get a prompt like this:

```
>
```

Try fetching a nonexistent file:

```
> fetch fooo
Couldn't find the file fooo
```

By starting several nodes (either on the same machine using different ports or on different machines) that know about each other (just put all the URLs in the URL files), you can try these out as you did with the first prototype. When you get bored with this, move on to the next section.

Further Exploration

You can probably think of several ways to improve and extend the system described in this chapter. Here are some ideas:

- Add caching. If your node relays a file through a call to `query`, why not store the file at the same time? That way, you can respond more quickly the next time someone asks for the same file. You could perhaps set a maximum size for the cache, remove old files, and so on.
- Use a threaded or asynchronous server (a bit difficult). That way, you can ask several other nodes for help without waiting for their replies, and they can later give you the reply by calling a `reply` method.
- Allow more advanced queries, such as querying on the contents of text files.
- Use the `hello` method more extensively. When you discover a new peer (through a call to `hello`), why not introduce it to all the peers you know? Perhaps you can think of more clever ways of discovering new peers?

- Read up on the representational state transfer (REST) philosophy of distributed systems. REST is an emerging alternative to web service technologies such as XML-RPC. (See, for example, `http://en.wikipedia.org/wiki/REST`.)
- Use `xmlrpclib.Binary` to wrap the files, to make the transfer safer for nontext files.
- Read the `SimpleXMLRPCServer` code. Check out the `DocXMLRPCServer` class and the multicall extension in `libxmlrpc`.

What Now?

Now that you have a peer-to-peer file sharing system working, how about making it more user friendly? In the next chapter, you learn how to add a GUI as an alternative to the current `cmd`-based interface.