CHAPTER 11

Network and Web Programming

This chapter is about various topics related to using Python in networked and distributed applications. Topics are split between using Python as a client to access existing services and using Python to implement networked services as a server. Common techniques for writing code involving cooperating or communicating with interpreters are also given.

11.1. Interacting with HTTP Services As a Client

Problem

You need to access various services via HTTP as a client. For example, downloading data or interacting with a REST-based API.

Solution

For simple things, it's usually easy enough to use the `urllib.request` module. For example, to send a simple HTTP `GET` request to a remote service, do something like this:

```
from urllib import request, parse

# Base URL being accessed
url = 'http://httpbin.org/get'

# Dictionary of query parameters (if any)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Encode the query string
querystring = parse.urlencode(parms)
```

```
# Make a GET request and read the response
u = request.urlopen(url+'?' + querystring)
resp = u.read()
```

If you need to send the query parameters in the request body using a `POST` method, encode them and supply them as an optional argument to `urlopen()` like this:

```
from urllib import request, parse

# Base URL being accessed
url = 'http://httpbin.org/post'

# Dictionary of query parameters (if any)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Encode the query string
querystring = parse.urlencode(parms)

# Make a POST request and read the response
u = request.urlopen(url, querystring.encode('ascii'))
resp = u.read()
```

If you need to supply some custom HTTP headers in the outgoing request such as a change to the user-agent field, make a dictionary containing their value and create a `Request` instance and pass it to `urlopen()` like this:

```
from urllib import request, parse
...

# Extra headers
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

req = request.Request(url, querystring.encode('ascii'), headers=headers)

# Make a request and read the response
u = request.urlopen(req)
resp = u.read()
```

If your interaction with a service is more complicated than this, you should probably look at the `requests` library. For example, here is equivalent `requests` code for the preceding operations:

```
import requests

# Base URL being accessed
url = 'http://httpbin.org/post'
```

```
# Dictionary of query parameters (if any)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Extra headers
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

resp = requests.post(url, data=parms, headers=headers)

# Decoded text returned by the request
text = resp.text
```

A notable feature of `requests` is how it returns the resulting response content from a request. As shown, the `resp.text` attribute gives you the Unicode decoded text of a request. However, if you access `resp.content`, you get the raw binary content instead. On the other hand, if you access `resp.json`, then you get the response content interpreted as JSON.

Here is an example of using `requests` to make a `HEAD` request and extract a few fields of header data from the response:

```
import requests

resp = requests.head('http://www.python.org/index.html')

status = resp.status_code
last_modified = resp.headers['last-modified']
content_type = resp.headers['content-type']
content_length = resp.headers['content-length']
```

Here is a `requests` example that executes a login into the Python Package index using basic authentication:

```
import requests

resp = requests.get('http://pypi.python.org/pypi?:action=login',
                    auth=('user','password'))
```

Here is an example of using `requests` to pass HTTP cookies from one request to the next:

```
import requests

# First request
resp1 = requests.get(url)
...
```

```
# Second requests with cookies received on first requests
resp2 = requests.get(url, cookies=resp1.cookies)
```

Last, but not least, here is an example of using `requests` to upload content:

```
import requests
url = 'http://httpbin.org/post'
files = { 'file': ('data.csv', open('data.csv', 'rb')) }

r = requests.post(url, files=files)
```

Discussion

For really simple HTTP client code, using the built-in `urllib` module is usually fine. However, if you have to do anything other than simple `GET` or `POST` requests, you really can't rely on its functionality. This is where a third-party module, such as `requests`, comes in handy.

For example, if you decided to stick entirely with the standard library instead of a library like `requests`, you might have to implement your code using the low-level `http.client` module instead. For example, this code shows how to execute a `HEAD` request:

```
from http.client import HTTPConnection
from urllib import parse

c = HTTPConnection('www.python.org', 80)
c.request('HEAD', '/index.html')
resp = c.getresponse()

print('Status', resp.status)
for name, value in resp.getheaders():
    print(name, value)
```

Similarly, if you have to write code involving proxies, authentication, cookies, and other details, using `urllib` is awkward and verbose. For example, here is a sample of code that authenticates to the Python package index:

```
import urllib.request

auth = urllib.request.HTTPBasicAuthHandler()
auth.add_password('pypi','http://pypi.python.org','username','password')
opener = urllib.request.build_opener(auth)

r = urllib.request.Request('http://pypi.python.org/pypi?:action=login')
u = opener.open(r)
resp = u.read()

# From here. You can access more pages using opener
...
```

Frankly, all of this is much easier in `requests`.

Testing HTTP client code during development can often be frustrating because of all the tricky details you need to worry about (e.g., cookies, authentication, headers, encodings, etc.). To do this, consider using the httpbin service. This site receives requests and then echoes information back to you in the form a JSON response. Here is an interactive example:

```
>>> import requests
>>> r = requests.get('http://httpbin.org/get?name=Dave&n=37',
...     headers = { 'User-agent': 'goaway/1.0' })
>>> resp = r.json
>>> resp['headers']
{'User-Agent': 'goaway/1.0', 'Content-Length': '', 'Content-Type': '',
'Accept-Encoding': 'gzip, deflate, compress', 'Connection':
'keep-alive', 'Host': 'httpbin.org', 'Accept': '*/*'}
>>> resp['args']
{'name': 'Dave', 'n': '37'}
>>>
```

Working with a site such as *httpbin.org* is often preferable to experimenting with a real site—especially if there's a risk it might shut down your account after three failed login attempts (i.e., don't try to learn how to write an HTTP authentication client by logging into your bank).

Although it's not discussed here, `requests` provides support for many more advanced HTTP-client protocols, such as OAuth. The `requests` documentation is excellent (and frankly better than anything that could be provided in this short space). Go there for more information.

11.2. Creating a TCP Server

Problem

You want to implement a server that communicates with clients using the TCP Internet protocol.

Solution

An easy way to create a TCP server is to use the `socketserver` library. For example, here is a simple echo server:

```
from socketserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        while True:
```

```
            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

In this code, you define a special handler class that implements a `handle()` method for servicing client connections. The `request` attribute is the underlying client socket and `client_address` has client address.

To test the server, run it and then open a separate Python process that connects to it:

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect(('localhost', 20000))
>>> s.send(b'Hello')
5
>>> s.recv(8192)
b'Hello'
>>>
```

In many cases, it may be easier to define a slightly different kind of handler. Here is an example that uses the `StreamRequestHandler` base class to put a file-like interface on the underlying socket:

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # self.rfile is a file-like object for reading
        for line in self.rfile:
            # self.wfile is a file-like object for writing
            self.wfile.write(line)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

Discussion

`socketserver` makes it relatively easy to create simple TCP servers. However, you should be aware that, by default, the servers are single threaded and can only serve one client at a time. If you want to handle multiple clients, either instantiate a `ForkingTCPServer` or `ThreadingTCPServer` object instead. For example:

```
from socketserver import ThreadingTCPServer
...
```

```
if __name__ == '__main__':
    serv = ThreadingTCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

One issue with forking and threaded servers is that they spawn a new process or thread on each client connection. There is no upper bound on the number of allowed clients, so a malicious hacker could potentially launch a large number of simultaneous connections in an effort to make your server explode.

If this is a concern, you can create a pre-allocated pool of worker threads or processes. To do this, you create an instance of a normal nonthreaded server, but then launch the `serve_forever()` method in a pool of multiple threads. For example:

```
...
if __name__ == '__main__':
    from threading import Thread
    NWORKERS = 16
    serv = TCPServer(('', 20000), EchoHandler)
    for n in range(NWORKERS):
        t = Thread(target=serv.serve_forever)
        t.daemon = True
        t.start()
    serv.serve_forever()
```

Normally, a `TCPServer` binds and activates the underlying socket upon instantiation. However, sometimes you might want to adjust the underlying socket by setting options. To do this, supply the `bind_and_activate=False` argument, like this:

```
if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler, bind_and_activate=False)
    # Set up various socket options
    serv.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    # Bind and activate
    serv.server_bind()
    serv.server_activate()
    serv.serve_forever()
```

The socket option shown is actually a very common setting that allows the server to rebind to a previously used port number. It's actually so common that it's a class variable that can be set on `TCPServer`. Set it before instantiating the server, as shown in this example:

```
...
if __name__ == '__main__':
    TCPServer.allow_reuse_address = True
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

In the solution, two different handler base classes were shown (`BaseRequestHandler` and `StreamRequestHandler`). The `StreamRequestHandler` class is actually a bit more

flexible, and supports some features that can be enabled through the specification of additional class variables. For example:

```
import socket

class EchoHandler(StreamRequestHandler):
    # Optional settings (defaults shown)
    timeout = 5                     # Timeout on all socket operations
    rbufsize = -1                   # Read buffer size
    wbufsize = 0                    # Write buffer size
    disable_nagle_algorithm = False  # Sets TCP_NODELAY socket option
    def handle(self):
        print('Got connection from', self.client_address)
        try:
            for line in self.rfile:
                # self.wfile is a file-like object for writing
                self.wfile.write(line)
        except socket.timeout:
            print('Timed out!')
```

Finally, it should be noted that most of Python's higher-level networking modules (e.g., HTTP, XML-RPC, etc.) are built on top of the `socketserver` functionality. That said, it is also not difficult to implement servers directly using the `socket` library as well. Here is a simple example of directly programming a server with Sockets:

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_handler(address, client_sock):
    print('Got connection from {}'.format(address))
    while True:
        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)
    client_sock.close()

def echo_server(address, backlog=5):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(backlog)
    while True:
        client_sock, client_addr = sock.accept()
        echo_handler(client_addr, client_sock)

if __name__ == '__main__':
    echo_server(('', 20000))
```

11.3. Creating a UDP Server

Problem

You want to implement a server that communicates with clients using the UDP Internet protocol.

Solution

As with TCP, UDP servers are also easy to create using the `socketserver` library. For example, here is a simple time server:

```
from socketserver import BaseRequestHandler, UDPServer
import time

class TimeHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # Get message and client socket
        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    serv = UDPServer(('', 20000), TimeHandler)
    serv.serve_forever()
```

As before, you define a special handler class that implements a `handle()` method for servicing client connections. The `request` attribute is a tuple that contains the incoming datagram and underlying socket object for the server. The `client_address` contains the client address.

To test the server, run it and then open a separate Python process that sends messages to it:

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 20000))
0
>>> s.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 20000))
>>>
```

Discussion

A typical UDP server receives an incoming datagram (message) along with a client address. If the server is to respond, it sends a datagram back to the client. For transmission of datagrams, you should use the `sendto()` and `recvfrom()` methods of a

socket. Although the traditional `send()` and `recv()` methods also might work, the former two methods are more commonly used with UDP communication.

Given that there is no underlying connection, UDP servers are often much easier to write than a TCP server. However, UDP is also inherently unreliable (e.g., no "connection" is established and messages might be lost). Thus, it would be up to you to figure out how to deal with lost messages. That's a topic beyond the scope of this book, but typically you might need to introduce sequence numbers, retries, timeouts, and other mechanisms to ensure reliability if it matters for your application. UDP is often used in cases where the requirement of reliable delivery can be relaxed. For instance, in real-time applications such as multimedia streaming and games where there is simply no option to go back in time and recover a lost packet (the program simply skips it and keeps moving forward).

The `UDPServer` class is single threaded, which means that only one request can be serviced at a time. In practice, this is less of an issue with UDP than with TCP connections. However, should you want concurrent operation, instantiate a `ForkingUDPServer` or `ThreadingUDPServer` object instead:

```
from socketserver import ThreadingUDPServer
...
if __name__ == '__main__':
    serv = ThreadingUDPServer(('',20000), TimeHandler)
    serv.serve_forever()
```

Implementing a UDP server directly using sockets is also not difficult. Here is an example:

```
from socket import socket, AF_INET, SOCK_DGRAM
import time

def time_server(address):
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.bind(address)
    while True:
        msg, addr = sock.recvfrom(8192)
        print('Got message from', addr)
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), addr)

if __name__ == '__main__':
    time_server(('', 20000))
```

11.4. Generating a Range of IP Addresses from a CIDR Address

Problem

You have a CIDR network address such as "123.45.67.89/27," and you want to generate a range of all the IP addresses that it represents (e.g., "123.45.67.64," "123.45.67.65," …, "123.45.67.95").

Solution

The `ipaddress` module can be easily used to perform such calculations. For example:

```
>>> import ipaddress
>>> net = ipaddress.ip_network('123.45.67.64/27')
>>> net
IPv4Network('123.45.67.64/27')
>>> for a in net:
...     print(a)
...
123.45.67.64
123.45.67.65
123.45.67.66
123.45.67.67
123.45.67.68
...
123.45.67.95
>>>

>>> net6 = ipaddress.ip_network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> net6
IPv6Network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> for a in net6:
...     print(a)
...
12:3456:78:90ab:cd:ef01:23:30
12:3456:78:90ab:cd:ef01:23:31
12:3456:78:90ab:cd:ef01:23:32
12:3456:78:90ab:cd:ef01:23:33
12:3456:78:90ab:cd:ef01:23:34
12:3456:78:90ab:cd:ef01:23:35
12:3456:78:90ab:cd:ef01:23:36
12:3456:78:90ab:cd:ef01:23:37
>>>
```

Network objects also allow indexing like arrays. For example:

```
>>> net.num_addresses
32
>>> net[0]
```

```
IPv4Address('123.45.67.64')
>>> net[1]
IPv4Address('123.45.67.65')
>>> net[-1]
IPv4Address('123.45.67.95')
>>> net[-2]
IPv4Address('123.45.67.94')
>>>
```

In addition, you can perform operations such as a check for network membership:

```
>>> a = ipaddress.ip_address('123.45.67.69')
>>> a in net
True
>>> b = ipaddress.ip_address('123.45.67.123')
>>> b in net
False
>>>
```

An IP address and network address can be specified together as an IP interface. For example:

```
>>> inet = ipaddress.ip_interface('123.45.67.73/27')
>>> inet.network
IPv4Network('123.45.67.64/27')
>>> inet.ip
IPv4Address('123.45.67.73')
>>>
```

Discussion

The `ipaddress` module has classes for representing IP addresses, networks, and interfaces. This can be especially useful if you want to write code that needs to manipulate network addresses in some way (e.g., parsing, printing, validating, etc.).

Be aware that there is only limited interaction between the `ipaddress` module and other network-related modules, such as the `socket` library. In particular, it is usually not possible to use an instance of `IPv4Address` as a substitute for address string. Instead, you have to explicitly convert it using `str()` first. For example:

```
>>> a = ipaddress.ip_address('127.0.0.1')
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect((a, 8080))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'IPv4Address' object to str implicitly
>>> s.connect((str(a), 8080))
>>>
```

See "An Introduction to the ipaddress Module" for more information and advanced usage.

11.5. Creating a Simple REST-Based Interface

Problem

You want to be able to control or interact with your program remotely over the network using a simple REST-based interface. However, you don't want to do it by installing a full-fledged web programming framework.

Solution

One of the easiest ways to build REST-based interfaces is to create a tiny library based on the WSGI standard, as described in PEP 3333. Here is an example:

```
# resty.py

import cgi

def notfound_404(environ, start_response):
    start_response('404 Not Found', [ ('Content-type', 'text/plain') ])
    return [b'Not Found']

class PathDispatcher:
    def __init__(self):
        self.pathmap = { }

    def __call__(self, environ, start_response):
        path = environ['PATH_INFO']
        params = cgi.FieldStorage(environ['wsgi.input'],
                                  environ=environ)
        method = environ['REQUEST_METHOD'].lower()
        environ['params'] = { key: params.getvalue(key) for key in params }
        handler = self.pathmap.get((method,path), notfound_404)
        return handler(environ, start_response)

    def register(self, method, path, function):
        self.pathmap[method.lower(), path] = function
        return function
```

To use this dispatcher, you simply write different handlers, such as the following:

```
import time

_hello_resp = '''\
<html>
  <head>
     <title>Hello {name}</title>
   </head>
   <body>
     <h1>Hello {name}!</h1>
   </body>
</html>'''
```

```
def hello_world(environ, start_response):
    start_response('200 OK', [ ('Content-type','text/html')])
    params = environ['params']
    resp = _hello_resp.format(name=params.get('name'))
    yield resp.encode('utf-8')

_localtime_resp = '''\
<?xml version="1.0"?>
<time>
  <year>{t.tm_year}</year>
  <month>{t.tm_mon}</month>
  <day>{t.tm_mday}</day>
  <hour>{t.tm_hour}</hour>
  <minute>{t.tm_min}</minute>
  <second>{t.tm_sec}</second>
</time>'''

def localtime(environ, start_response):
    start_response('200 OK', [ ('Content-type', 'application/xml') ])
    resp = _localtime_resp.format(t=time.localtime())
    yield resp.encode('utf-8')

if __name__ == '__main__':
    from resty import PathDispatcher
    from wsgiref.simple_server import make_server

    # Create the dispatcher and register functions
    dispatcher = PathDispatcher()
    dispatcher.register('GET', '/hello', hello_world)
    dispatcher.register('GET', '/localtime', localtime)

    # Launch a basic server
    httpd = make_server('', 8080, dispatcher)
    print('Serving on port 8080...')
    httpd.serve_forever()
```

To test your server, you can interact with it using a browser or `urllib`. For example:

```
>>> u = urlopen('http://localhost:8080/hello?name=Guido')
>>> print(u.read().decode('utf-8'))
<html>
  <head>
     <title>Hello Guido</title>
   </head>
   <body>
     <h1>Hello Guido!</h1>
   </body>
</html>
>>> u = urlopen('http://localhost:8080/localtime')
>>> print(u.read().decode('utf-8'))
<?xml version="1.0"?>
<time>
```

```
  <year>2012</year>
  <month>11</month>
  <day>24</day>
  <hour>14</hour>
  <minute>49</minute>
  <second>17</second>
</time>
>>>
```

Discussion

In REST-based interfaces, you are typically writing programs that respond to common HTTP requests. However, unlike a full-fledged website, you're often just pushing data around. This data might be encoded in a variety of standard formats such as XML, JSON, or CSV. Although it seems minimal, providing an API in this manner can be a very useful thing for a wide variety of applications.

For example, long-running programs might use a REST API to implement monitoring or diagnostics. Big data applications can use REST to build a query/data extraction system. REST can even be used to control hardware devices, such as robots, sensors, mills, or lightbulbs. What's more, REST APIs are well supported by various client-side programming environments, such as Javascript, Android, iOS, and so forth. Thus, having such an interface can be a way to encourage the development of more complex applications that interface with your code.

For implementing a simple REST interface, it is often easy enough to base your code on the Python WSGI standard. WSGI is supported by the standard library, but also by most third-party web frameworks. Thus, if you use it, there is a lot of flexibility in how your code might be used later.

In WSGI, you simply implement applications in the form of a callable that accepts this calling convention:

```
import cgi

def wsgi_app(environ, start_response):
    ...
```

The `environ` argument is a dictionary that contains values inspired by the CGI interface provided by various web servers such as Apache [see Internet RFC 3875]. To extract different fields, you would write code like this:

```
def wsgi_app(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ['PATH_INFO']
    # Parse the query parameters
    params = cgi.FieldStorage(environ['wsgi.input'], environ=environ)
    ...
```

A few common values are shown here. `environ['REQUEST_METHOD']` is the type of request (e.g., `GET`, `POST`, `HEAD`, etc.). `environ['PATH_INFO']` is the path or the resource being requested. The call to `cgi.FieldStorage()` extracts supplied query parameters from the request and puts them into a dictionary-like object for later use.

The `start_response` argument is a function that must be called to initiate a response. The first argument is the resulting HTTP status. The second argument is a list of `(name, value)` tuples that make up the HTTP headers of the response. For example:

```
def wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
```

To return data, an WSGI application must return a sequence of byte strings. This can be done using a list like this:

```
def wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
    resp = []
    resp.append(b'Hello World\n')
    resp.append(b'Goodbye!\n')
    return resp
```

Alternatively, you can use `yield`:

```
def wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
    yield b'Hello World\n'
    yield b'Goodbye!\n'
```

It's important to emphasize that byte strings must be used in the result. If the response consists of text, it will need to be encoded into bytes first. Of course, there is no requirement that the returned value be text—you could easily write an application function that creates images.

Although WSGI applications are commonly defined as a function, as shown, an instance may also be used as long as it implements a suitable `__call__()` method. For example:

```
class WSGIApplication:
    def __init__(self):
        ...
    def __call__(self, environ, start_response)
       ...
```

This technique has been used to create the `PathDispatcher` class in the recipe. The dispatcher does nothing more than manage a dictionary mapping `(method, path)` pairs to handler functions. When a request arrives, the method and path are extracted and used to dispatch to a handler. In addition, any query variables are parsed and put into

a dictionary that is stored as `environ['params']` (this latter step is so common, it makes a lot of sense to simply do it in the dispatcher in order to avoid a lot of replicated code).

To use the dispatcher, you simply create an instance and register various WSGI-style application functions with it, as shown in the recipe. Writing these functions should be extremely straightforward, as you follow the rules concerning the `start_response()` function and produce output as byte strings.

One thing to consider when writing such functions is the careful use of string templates. Nobody likes to work with code that is a tangled mess of `print()` functions, XML, and various formatting operations. In the solution, triple-quoted string templates are being defined and used internally. This particular approach makes it easier to change the format of the output later (just change the template as opposed to any of the code that uses it).

Finally, an important part of using WSGI is that nothing in the implementation is specific to a particular web server. That is actually the whole idea—since the standard is server and framework neutral, you should be able to plug your application into a wide variety of servers. In the recipe, the following code is used for testing:

```
if __name__ == '__main__':
    from wsgiref.simple_server import make_server

    # Create the dispatcher and register functions
    dispatcher = PathDispatcher()
    ...

    # Launch a basic server
    httpd = make_server('', 8080, dispatcher)
    print('Serving on port 8080...')
    httpd.serve_forever()
```

This will create a simple server that you can use to see if your implementation works. Later on, when you're ready to scale things up to a larger level, you will change this code to work with a particular server.

WSGI is an intentionally minimal specification. As such, it doesn't provide any support for more advanced concepts such as authentication, cookies, redirection, and so forth. These are not hard to implement yourself. However, if you want just a bit more support, you might consider third-party libraries, such as WebOb or Paste.

11.6. Implementing a Simple Remote Procedure Call with XML-RPC

Problem

You want an easy way to execute functions or methods in Python programs running on remote machines.

Solution

Perhaps the easiest way to implement a simple remote procedure call mechanism is to use XML-RPC. Here is an example of a simple server that implements a simple key-value store:

```
from xmlrpc.server import SimpleXMLRPCServer

class KeyValueServer:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def __init__(self, address):
        self._data = {}
        self._serv = SimpleXMLRPCServer(address, allow_none=True)
        for name in self._rpc_methods_:
            self._serv.register_function(getattr(self, name))

    def get(self, name):
        return self._data[name]

    def set(self, name, value):
        self._data[name] = value

    def delete(self, name):
        del self._data[name]

    def exists(self, name):
        return name in self._data

    def keys(self):
        return list(self._data)

    def serve_forever(self):
        self._serv.serve_forever()

# Example
if __name__ == '__main__':
    kvserv = KeyValueServer(('', 15000))
    kvserv.serve_forever()
```

Here is how you would access the server remotely from a client:

```
>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy('http://localhost:15000', allow_none=True)
>>> s.set('foo', 'bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>
```

Discussion

XML-RPC can be an extremely easy way to set up a simple remote procedure call service. All you need to do is create a server instance, register functions with it using the `register_function()` method, and then launch it using the `serve_forever()` method. This recipe packages it up into a class to put all of the code together, but there is no such requirement. For example, you could create a server by trying something like this:

```
from xmlrpc.server import SimpleXMLRPCServer
def add(x,y):
    return x+y

serv = SimpleXMLRPCServer(('', 15000))
serv.register_function(add)
serv.serve_forever()
```

Functions exposed via XML-RPC only work with certain kinds of data such as strings, numbers, lists, and dictionaries. For everything else, some study is required. For instance, if you pass an instance through XML-RPC, only its instance dictionary is handled:

```
>>> class Point:
...     def __init__(self, x, y):
...             self.x = x
...             self.y = y
...
>>> p = Point(2, 3)
>>> s.set('foo', p)
>>> s.get('foo')
{'x': 2, 'y': 3}
>>>
```

Similarly, handling of binary data is a bit different than you expect:

```
>>> s.set('foo', b'Hello World')
>>> s.get('foo')
<xmlrpc.client.Binary object at 0x10131d410>
```

```
>>> _.data
b'Hello World'
>>>
```

As a general rule, you probably shouldn't expose an XML-RPC service to the rest of the world as a public API. It often works best on internal networks where you might want to write simple distributed programs involving a few different machines.

A downside to XML-RPC is its performance. The `SimpleXMLRPCServer` implementation is only single threaded, and wouldn't be appropriate for scaling a large application, although it can be made to run multithreaded, as shown in Recipe 11.2. Also, since XML-RPC serializes all data as XML, it's inherently slower than other approaches. However, one benefit of this encoding is that it's understood by a variety of other programming languages. By using it, clients written in languages other than Python will be able to access your service.

Despite its limitations, XML-RPC is worth knowing about if you ever have the need to make a quick and dirty remote procedure call system. Oftentimes, the simple solution is good enough.

11.7. Communicating Simply Between Interpreters

Problem

You are running multiple instances of the Python interpreter, possibly on different machines, and you would like to exchange data between interpreters using messages.

Solution

It is easy to communicate between interpreters if you use the `multiprocessing.connection` module. Here is a simple example of writing an echo server:

```
from multiprocessing.connection import Listener
import traceback

def echo_client(conn):
    try:
        while True:
            msg = conn.recv()
            conn.send(msg)
    except EOFError:
        print('Connection closed')

def echo_server(address, authkey):
    serv = Listener(address, authkey=authkey)
    while True:
        try:
            client = serv.accept()
```

```
            echo_client(client)
        except Exception:
            traceback.print_exc()

echo_server(('', 25000), authkey=b'peekaboo')
```

Here is a simple example of a client connecting to the server and sending various messages:

```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 25000), authkey=b'peekaboo')
>>> c.send('hello')
>>> c.recv()
'hello'
>>> c.send(42)
>>> c.recv()
42
>>> c.send([1, 2, 3, 4, 5])
>>> c.recv()
[1, 2, 3, 4, 5]
>>>
```

Unlike a low-level socket, messages are kept intact (each object sent using `send()` is received in its entirety with `recv()`). In addition, objects are serialized using `pickle`. So, any object compatible with `pickle` can be sent or received over the connection.

Discussion

There are many packages and libraries related to implementing various forms of message passing, such as ZeroMQ, Celery, and so forth. As an alternative, you might also be inclined to implement a message layer on top of low-level sockets. However, sometimes you just want a simple solution. The `multiprocessing.connection` library is just that—using a few simple primitives, you can easily connect interpreters together and have them exchange messages.

If you know that the interpreters are going to be running on the same machine, you can use alternative forms of networking, such as UNIX domain sockets or Windows named pipes. To create a connection using a UNIX domain socket, simply change the address to a filename such as this:

```
s = Listener('/tmp/myconn', authkey=b'peekaboo')
```

To create a connection using a Windows named pipe, use a filename such as this:

```
s = Listener(r'\\.\pipe\myconn', authkey=b'peekaboo')
```

As a general rule, you would not be using `multiprocessing` to implement public-facing services. The `authkey` parameter to `Client()` and `Listener()` is there to help authenticate the end points of the connection. Connection attempts with a bad key raise an exception. In addition, the module is probably best suited for long-running connections

(not a large number of short connections). For example, two interpreters might establish a connection at startup and keep the connection active for the entire duration of a problem.

Don't use `multiprocessing` if you need more low-level control over aspects of the connection. For example, if you needed to support timeouts, nonblocking I/O, or anything similar, you're probably better off using a different library or implementing such features on top of sockets instead.

11.8. Implementing Remote Procedure Calls

Problem

You want to implement simple remote procedure call (RPC) on top of a message passing layer, such as sockets, multiprocessing connections, or ZeroMQ.

Solution

RPC is easy to implement by encoding function requests, arguments, and return values using `pickle`, and passing the pickled byte strings between interpreters. Here is an example of a simple RPC handler that could be incorporated into a server:

```
# rpcserver.py

import pickle
class RPCHandler:
    def __init__(self):
        self._functions = { }

    def register_function(self, func):
        self._functions[func.__name__] = func

    def handle_connection(self, connection):
        try:
            while True:
                # Receive a message
                func_name, args, kwargs = pickle.loads(connection.recv())
                # Run the RPC and send a response
                try:
                    r = self._functions[func_name](*args,**kwargs)
                    connection.send(pickle.dumps(r))
                except Exception as e:
                    connection.send(pickle.dumps(e))
        except EOFError:
             pass
```

To use this handler, you need to add it into a messaging server. There are many possible choices, but the `multiprocessing` library provides a simple option. Here is an example RPC server:

```
from multiprocessing.connection import Listener
from threading import Thread

def rpc_server(handler, address, authkey):
    sock = Listener(address, authkey=authkey)
    while True:
        client = sock.accept()
        t = Thread(target=handler.handle_connection, args=(client,))
        t.daemon = True
        t.start()

# Some remote functions
def add(x, y):
    return x + y

def sub(x, y):
    return x - y

# Register with a handler
handler = RPCHandler()
handler.register_function(add)
handler.register_function(sub)

# Run the server
rpc_server(handler, ('localhost', 17000), authkey=b'peekaboo')
```

To access the server from a remote client, you need to create a corresponding RPC proxy class that forwards requests. For example:

```
import pickle

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(pickle.dumps((name, args, kwargs)))
            result = pickle.loads(self._connection.recv())
            if isinstance(result, Exception):
                raise result
            return result
        return do_rpc
```

To use the proxy, you wrap it around a connection to the server. For example:

```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 17000), authkey=b'peekaboo')
>>> proxy = RPCProxy(c)
>>> proxy.add(2, 3)
```

```
5
>>> proxy.sub(2, 3)
-1
>>> proxy.sub([1, 2], 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "rpcserver.py", line 37, in do_rpc
    raise result
TypeError: unsupported operand type(s) for -: 'list' and 'int'
>>>
```

It should be noted that many messaging layers (such as `multiprocessing`) already serialize data using `pickle`. If this is the case, the `pickle.dumps()` and `pickle.loads()` calls can be eliminated.

Discussion

The general idea of the `RPCHandler` and `RPCProxy` classes is relatively simple. If a client wants to call a remote function, such as `foo(1, 2, z=3)`, the proxy class creates a tuple `('foo', (1, 2), {'z': 3})` that contains the function name and arguments. This tuple is pickled and sent over the connection. This is performed in the `do_rpc()` closure that's returned by the `__getattr__()` method of `RPCProxy`. The server receives and unpickles the message, looks up the function name to see if it's registered, and executes it with the given arguments. The result (or exception) is then pickled and sent back.

As shown, the example relies on `multiprocessing` for communication. However, this approach could be made to work with just about any other messaging system. For example, if you want to implement RPC over ZeroMQ, just replace the connection objects with an appropriate ZeroMQ socket object.

Given the reliance on `pickle`, security is a major concern (because a clever hacker can create messages that make arbitrary functions execute during unpickling). In particular, you should never allow RPC from untrusted or unauthenticated clients. In particular, you definitely don't want to allow access from just any machine on the Internet—this should really only be used internally, behind a firewall, and not exposed to the rest of the world.

As an alternative to `pickle`, you might consider the use of JSON, XML, or some other data encoding for serialization. For example, this recipe is fairly easy to adapt to JSON encoding if you simply replace `pickle.loads()` and `pickle.dumps()` with `json.loads()` and `json.dumps()`. For example:

```
# jsonrpcserver.py
import json

class RPCHandler:
    def __init__(self):
        self._functions = { }
```

```
    def register_function(self, func):
        self._functions[func.__name__] = func

    def handle_connection(self, connection):
        try:
            while True:
                # Receive a message
                func_name, args, kwargs = json.loads(connection.recv())
                # Run the RPC and send a response
                try:
                    r = self._functions[func_name](*args,**kwargs)
                    connection.send(json.dumps(r))
                except Exception as e:
                    connection.send(json.dumps(str(e)))
        except EOFError:
             pass

# jsonrpcclient.py
import json

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(json.dumps((name, args, kwargs)))
            result = json.loads(self._connection.recv())
            return result
        return do_rpc
```

One complicated factor in implementing RPC is how to handle exceptions. At the very least, the server shouldn't crash if an exception is raised by a method. However, the means by which the exception gets reported back to the client requires some study. If you're using `pickle`, exception instances can often be serialized and reraised in the client. If you're using some other protocol, you might have to think of an alternative approach. At the very least, you would probably want to return the exception string in the response. This is the approach taken in the JSON example.

For another example of an RPC implementation, it can be useful to look at the implementation of the `SimpleXMLRPCServer` and `ServerProxy` classes used in XML-RPC, as described in Recipe 11.6.

11.9. Authenticating Clients Simply

Problem

You want a simple way to authenticate the clients connecting to servers in a distributed system, but don't need the complexity of something like SSL.

Solution

Simple but effective authentication can be performed by implementing a connection handshake using the `hmac` module. Here is sample code:

```
import hmac
import os

def client_authenticate(connection, secret_key):
    '''
    Authenticate client to a remote service.
    connection represents a network connection.
    secret_key is a key known only to both client/server.
    '''
    message = connection.recv(32)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    connection.send(digest)

def server_authenticate(connection, secret_key):
    '''
    Request client authentication.
    '''
    message = os.urandom(32)
    connection.send(message)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    response = connection.recv(len(digest))
    return hmac.compare_digest(digest,response)
```

The general idea is that upon connection, the server presents the client with a message of random bytes (returned by `os.urandom()`, in this case). The client and server both compute a cryptographic hash of the random data using `hmac` and a secret key known only to both ends. The client sends its computed digest back to the server, where it is compared and used to decide whether or not to accept or reject the connection.

Comparison of resulting digests should be performed using the `hmac.compare_digest()` function. This function has been written in a way that avoids timing-analysis-based attacks and should be used instead of a normal comparison operator (`==`).

To use these functions, you would incorporate them into existing networking or messaging code. For example, with sockets, the server code might look something like this:

```
from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'peekaboo'
def echo_handler(client_sock):
    if not server_authenticate(client_sock, secret_key):
        client_sock.close()
        return
    while True:
```

```
            msg = client_sock.recv(8192)
            if not msg:
                break
            client_sock.sendall(msg)

    def echo_server(address):
        s = socket(AF_INET, SOCK_STREAM)
        s.bind(address)
        s.listen(5)
        while True:
            c,a = s.accept()
            echo_handler(c)

    echo_server(('', 18000))
```

Within a client, you would do this:

```
    from socket import socket, AF_INET, SOCK_STREAM

    secret_key = b'peekaboo'

    s = socket(AF_INET, SOCK_STREAM)
    s.connect(('localhost', 18000))
    client_authenticate(s, secret_key)
    s.send(b'Hello World')
    resp = s.recv(1024)
    ...
```

Discussion

A common use of `hmac` authentication is in internal messaging systems and interprocess communication. For example, if you are writing a system that involves multiple processes communicating across a cluster of machines, you can use this approach to make sure that only allowed processes are allowed to connect to one another. In fact, HMAC-based authentication is used internally by the `multiprocessing` library when it sets up communication with subprocesses.

It's important to stress that authenticating a connection is not the same as encryption. Subsequent communication on an authenticated connection is sent in the clear, and would be visible to anyone inclined to sniff the traffic (although the secret key known to both sides is never transmitted).

The authentication algorithm used by `hmac` is based on cryptographic hashing functions, such as MD5 and SHA-1, and is described in detail in IETF RFC 2104.

11.10. Adding SSL to Network Services

Problem

You want to implement a network service involving sockets where servers and clients authenticate themselves and encrypt the transmitted data using SSL.

Solution

The `ssl` module provides support for adding SSL to low-level socket connections. In particular, the `ssl.wrap_socket()` function takes an existing socket and wraps an SSL layer around it. For example, here's an example of a simple echo server that presents a server certificate to connecting clients:

```
from socket import socket, AF_INET, SOCK_STREAM
import ssl

KEYFILE = 'server_key.pem'   # Private key of the server
CERTFILE = 'server_cert.pem' # Server certificate (given to client)

def echo_client(s):
    while True:
        data = s.recv(8192)
        if data == b'':
            break
        s.send(data)
    s.close()
    print('Connection closed')

def echo_server(address):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
    s.listen(1)

    # Wrap with an SSL layer requiring client certs
    s_ssl = ssl.wrap_socket(s,
                            keyfile=KEYFILE,
                            certfile=CERTFILE,
                            server_side=True
                            )
    # Wait for connections
    while True:
        try:
            c,a = s_ssl.accept()
            print('Got connection', c, a)
            echo_client(c)
        except Exception as e:
            print('{}: {}'.format(e.__class__.__name__, e))

echo_server(('', 20000))
```

Here's an interactive session that shows how to connect to the server as a client. The client requires the server to present its certificate and verifies it:

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> import ssl
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s_ssl = ssl.wrap_socket(s,
...                         cert_reqs=ssl.CERT_REQUIRED,
...                         ca_certs = 'server_cert.pem')
>>> s_ssl.connect(('localhost', 20000))
>>> s_ssl.send(b'Hello World?')
12
>>> s_ssl.recv(8192)
b'Hello World?'
>>>
```

The problem with all of this low-level socket hacking is that it doesn't play well with existing network services already implemented in the standard library. For example, most server code (HTTP, XML-RPC, etc.) is actually based on the `socketserver` library. Client code is also implemented at a higher level. It is possible to add SSL to existing services, but a slightly different approach is needed.

First, for servers, SSL can be added through the use of a mixin class like this:

```
import ssl

class SSLMixin:
    '''
    Mixin class that adds support for SSL to existing servers based
    on the socketserver module.
    '''
    def __init__(self, *args,
                 keyfile=None, certfile=None, ca_certs=None,
                 cert_reqs=ssl.NONE,
                 **kwargs):
        self._keyfile = keyfile
        self._certfile = certfile
        self._ca_certs = ca_certs
        self._cert_reqs = cert_reqs
        super().__init__(*args, **kwargs)

    def get_request(self):
        client, addr = super().get_request()
        client_ssl = ssl.wrap_socket(client,
                                     keyfile = self._keyfile,
                                     certfile = self._certfile,
                                     ca_certs = self._ca_certs,
                                     cert_reqs = self._cert_reqs,
                                     server_side = True)
        return client_ssl, addr
```

To use this mixin class, you can mix it with other server classes. For example, here's an example of defining an XML-RPC server that operates over SSL:

```
# XML-RPC server with SSL

from xmlrpc.server import SimpleXMLRPCServer

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):
    pass
```

Here's the XML-RPC server from Recipe 11.6 modified only slightly to use SSL:

```
import ssl
from xmlrpc.server import SimpleXMLRPCServer
from sslmixin import SSLMixin

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):
    pass

class KeyValueServer:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def __init__(self, *args, **kwargs):
        self._data = {}
        self._serv = SSLSimpleXMLRPCServer(*args, allow_none=True, **kwargs)
        for name in self._rpc_methods_:
            self._serv.register_function(getattr(self, name))

    def get(self, name):
        return self._data[name]

    def set(self, name, value):
        self._data[name] = value

    def delete(self, name):
        del self._data[name]

    def exists(self, name):
        return name in self._data

    def keys(self):
        return list(self._data)

    def serve_forever(self):
        self._serv.serve_forever()

if __name__ == '__main__':
    KEYFILE='server_key.pem'    # Private key of the server
    CERTFILE='server_cert.pem'  # Server certificate
    kvserv = KeyValueServer(('', 15000),
                            keyfile=KEYFILE,
                            certfile=CERTFILE),
    kvserv.serve_forever()
```

To use this server, you can connect using the normal `xmlrpc.client` module. Just specify a `https:` in the URL. For example:

```
>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy('https://localhost:15000', allow_none=True)
>>> s.set('foo','bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>
```

One complicated issue with SSL clients is performing extra steps to verify the server certificate or to present a server with client credentials (such as a client certificate). Unfortunately, there seems to be no standardized way to accomplish this, so research is often required. However, here is an example of how to set up a secure XML-RPC connection that verifies the server's certificate:

```
from xmlrpc.client import SafeTransport, ServerProxy
import ssl

class VerifyCertSafeTransport(SafeTransport):
    def __init__(self, cafile, certfile=None, keyfile=None):
        SafeTransport.__init__(self)
        self._ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
        self._ssl_context.load_verify_locations(cafile)
        if cert:
            self._ssl_context.load_cert_chain(certfile, keyfile)
        self._ssl_context.verify_mode = ssl.CERT_REQUIRED

    def make_connection(self, host):
        # Items in the passed dictionary are passed as keyword
        # arguments to the http.client.HTTPSConnection() constructor.
        # The context argument allows an ssl.SSLContext instance to
        # be passed with information about the SSL configuration
        s = super().make_connection((host, {'context': self._ssl_context}))

        return s

# Create the client proxy
s = ServerProxy('https://localhost:15000',
                transport=VerifyCertSafeTransport('server_cert.pem'),
                allow_none=True)
```

As shown, the server presents a certificate to the client and the client verifies it. This verification can go both directions. If the server wants to verify the client, change the server startup to the following:

```
if __name__ == '__main__':
    KEYFILE='server_key.pem'   # Private key of the server
    CERTFILE='server_cert.pem' # Server certificate
    CA_CERTS='client_cert.pem' # Certificates of accepted clients

    kvserv = KeyValueServer(('', 15000),
                            keyfile=KEYFILE,
                            certfile=CERTFILE,
                            ca_certs=CA_CERTS,
                            cert_reqs=ssl.CERT_REQUIRED,
                            )
    kvserv.serve_forever()
```

To make the XML-RPC client present its certificates, change the `ServerProxy` initialization to this:

```
# Create the client proxy
s = ServerProxy('https://localhost:15000',
                transport=VerifyCertSafeTransport('server_cert.pem',
                                                  'client_cert.pem',
                                                  'client_key.pem'),
                allow_none=True)
```

Discussion

Getting this recipe to work will test your system configuration skills and understanding of SSL. Perhaps the biggest challenge is simply getting the initial configuration of keys, certificates, and other matters in order.

To clarify what's required, each endpoint of an SSL connection typically has a private key and a signed certificate file. The certificate file contains the public key and is presented to the remote peer on each connection. For public servers, certificates are normally signed by a certificate authority such as Verisign, Equifax, or similar organization (something that costs money). To verify server certificates, clients maintain a file containing the certificates of trusted certificate authorities. For example, web browsers maintain certificates corresponding to the major certificate authorities and use them to verify the integrity of certificates presented by web servers during HTTPS connections.

For the purposes of this recipe, you can create what's known as a self-signed certificate. Here's how you do it:

```
bash % openssl req -new -x509 -days 365 -nodes -out server_cert.pem \
           -keyout server_key.pem
Generating a 1024 bit RSA private key
.....................................++++++
...++++++
```

```
writing new private key to 'server_key.pem'
 -----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
 -----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Illinois
Locality Name (eg, city) []:Chicago
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Dabeaz, LLC
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []:localhost
Email Address []:
bash %
```

When creating the certificate, the values for the various fields are often arbitrary. However, the "Common Name" field often contains the DNS hostname of servers. If you're just testing things out on your own machine, use "localhost." Otherwise, use the domain name of the machine that's going to run the server.

As a result of this configuration, you will have a *server_key.pem* file that contains the private key. It looks like this:

```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQCZrCNLoEyAKF+f9UNcFaz5Osa6jf7qkbUl8si5xQrY3ZYC7juu
nL1dZLn/VbEFIITaUOgvBtPv1qUWTJGwga62VSG1oFE0ODIx3g2Nh4sRf+rySsx2
L4442nx0z4O5vJQ7k6eRNHAZUUnCL50+YvjyLyt7ryLSjSuKhCcJsbZgPwIDAQAB
AoGAB5evrr7eyL4160tM5rHTeATlaLY3UBOe5Z8XN8Z6gLiB/ucSX9AysviVD/6F
3oD6z2aL8jbeJc1vHqjt0dC2dwwm32vVl8mRdyoAsQpWmiqXrkvP4Bsl04VpBeHw
Qt8xNSW9SFhceL3LEvw9M8i9MV39viih1ILyH8OuHdvJyFECQQDLEjl2d2ppxND9
PoLqVFAirDfX2JnLTdWbc+M11a9Jdn3hKF8TcxfEnFVs5Gav1MusicY5KB0ylYPb
YbTvqKc7AkEAwbnRBO2VYEZsJZp2X0IZqP9ovWokkpYx+PE4+c6MySDgaMcigL7v
WDIHJG1CHudD09GbqENasDzyb2HAIW4CzQJBAKDdkv+xoW6gJx42Auc2WzTcUHCA
eXR/+BLpPrhKykzbvOQ8YvS5W764SUO1u1LWs3G+wnRMvrRvlMCZKgggBjkCQQCG
Jewto2+a+WkOKQXrNNScCDE5aPTmZQc5waCYq4UmCZQcOjkUOiN3ST1U5iuxRqfb
V/yX6fw0qh+fLWtkOs/JAkA+okMSxZwqRtfgOFGBfwQ8/iKrnizeanTQ3L6scFXI
CHZXdJ3XQ6qUmNxNn7iJ7S/LDawo1QfWkCfD9FYoxBlg
-----END RSA PRIVATE KEY-----
```

The server certificate in `server_cert.pem` looks similar:

```
-----BEGIN CERTIFICATE-----
MIIC+DCCAmGgAwIBAgIJAPMd+vi45js3MA0GCSqGSIb3DQEBBQUAMFwxCzAJBgNV
BAYTAlVTMREwDwYDVQQIEwhJbGxpbm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIG
A1UEChMLRGFiZWF6LCBMTEMxEjAQBgNVBAMTCWxvY2FsaG9zdDAeFw0xMzAxMTEx
ODQyMjdaFw0xNDAxMTExODQyMjdaMFwxCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhJ
bGxpbm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIGA1UEChMLRGFiZWF6LCBMTEMx
EjAQBgNVBAMTCWxvY2FsaG9zdDCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA
mawjS6BMgChfn/VDXBWs+TrGuo3+6pG1JfLIucUK2N2WAu47rpy9XWS5/1WxBSCE
2lDoLwbT79alFkyRsIGutlUhtaBRNDgyMd4NjYeLEX/q8krMdi+OONp8dM+DubyU
```

```
O5OnkTRwGVFJwi+dPmL48i8re68i0o0rioQnCbG2YD8CAwEAAaOBwTCBvjAdBgNV
HQ4EFgQUrtoLHHgXiDZTr26NMmgKJLJLFtIwgY4GA1UdIwSBhjCBg4AUrtoLHHgX
iDZTr26NMmgKJLJLFtKhYKReMFwxCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhJbGxp
bm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIGA1UEChMLRGFiZWF6LCBMTEMxEjAQ
BgNVBAMTCWxvY2FsaG9zdIIJAPMd+vi45js3MAwGA1UdEwQFMAMBAf8wDQYJKoZI
hvcNAQEFBQADgYEAFci+dqvMG4xF8UTnbGVvZJPIzJDRee6Nbt6AHQo9pOdAIMAu
WsGCplSOaDNdKKzl+b2UT2Zp3AIW4Qd51bouSNnR4M/gnr9ZD1ZctFd3jS+C5XRp
D3vvcW5lAnCCC80P6rXy7d7hTeFu5EYKtRGXNvVNd/06NALGDflrrOwxF3Y=
-----END CERTIFICATE-----
```

In server-related code, both the private key and certificate file will be presented to the various SSL-related wrapping functions. The certificate is what gets presented to clients. The private key should be protected and remains on the server.

In client-related code, a special file of valid certificate authorities needs to be maintained to verify the server's certificate. If you have no such file, then at the very least, you can put a copy of the server's certificate on the client machine and use that as a means for verification. During connection, the server will present its certificate, and then you'll use the stored certificate you already have to verify that it's correct.

Servers can also elect to verify the identity of clients. To do that, clients need to have their own private key and certificate key. The server would also need to maintain a file of trusted certificate authorities for verifying the client certificates.

If you intend to add SSL support to a network service for real, this recipe really only gives a small taste of how to set it up. You will definitely want to consult the documentation for more of the finer points. Be prepared to spend a significant amount of time experimenting with it to get things to work.

11.11. Passing a Socket File Descriptor Between Processes

Problem

You have multiple Python interpreter processes running and you want to pass an open file descriptor from one interpreter to the other. For instance, perhaps there is a server process that is responsible for receiving connections, but the actual servicing of clients is to be handled by a different interpreter.

Solution

To pass a file descriptor between processes, you first need to connect the processes together. On Unix machines, you might use a Unix domain socket, whereas on Windows, you could use a named pipe. However, rather than deal with such low-level mechanics, it is often easier to use the `multiprocessing` module to set up such a connection.

Once a connection is established, you can use the `send_handle()` and `recv_handle()` functions in `multiprocessing.reduction` to send file descriptors between processes.

The following example illustrates the basics:

```
import multiprocessing
from multiprocessing.reduction import recv_handle, send_handle
import socket

def worker(in_p, out_p):
    out_p.close()
    while True:
        fd = recv_handle(in_p)
        print('CHILD: GOT FD', fd)
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd) as s:
            while True:
                msg = s.recv(1024)
                if not msg:
                    break
                print('CHILD: RECV {!r}'.format(msg))
                s.send(msg)

def server(address, in_p, out_p, worker_pid):
    in_p.close()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(address)
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
        send_handle(out_p, client.fileno(), worker_pid)
        client.close()

if __name__ == '__main__':
    c1, c2 = multiprocessing.Pipe()
    worker_p = multiprocessing.Process(target=worker, args=(c1,c2))
    worker_p.start()

    server_p = multiprocessing.Process(target=server,
                 args=(('', 15000), c1, c2, worker_p.pid))
    server_p.start()

    c1.close()
    c2.close()
```

In this example, two processes are spawned and connected by a multiprocessing `Pipe` object. The server process opens a socket and waits for client connections. The worker process merely waits to receive a file descriptor on the pipe using `recv_handle()`. When the server receives a connection, it sends the resulting socket file descriptor to the worker

using `send_handle()`. The worker takes over the socket and echoes data back to the client until the connection is closed.

If you connect to the running server using Telnet or a similar tool, here is an example of what you might see:

```
bash % python3 passfd.py
SERVER: Got connection from ('127.0.0.1', 55543)
CHILD: GOT FD 7
CHILD: RECV b'Hello\r\n'
CHILD: RECV b'World\r\n'
```

The most important part of this example is the fact that the client socket accepted in the server is actually serviced by a completely different process. The server merely hands it off, closes it, and waits for the next connection.

Discussion

Passing file descriptors between processes is something that many programmers don't even realize is possible. However, it can sometimes be a useful tool in building scalable systems. For example, on a multicore machine, you could have multiple instances of the Python interpreter and use file descriptor passing to more evenly balance the number of clients being handled by each interpreter.

The `send_handle()` and `recv_handle()` functions shown in the solution really only work with multiprocessing connections. Instead of using a pipe, you can connect interpreters as shown in Recipe 11.7, and it will work as long as you use UNIX domain sockets or Windows pipes. For example, you could implement the server and worker as completely separate programs to be started separately. Here is the implementation of the server:

```
# servermp.py
from multiprocessing.connection import Listener
from multiprocessing.reduction import send_handle
import socket

def server(work_address, port):
    # Wait for the worker to connect
    work_serv = Listener(work_address, authkey=b'peekaboo')
    worker = work_serv.accept()
    worker_pid = worker.recv()

    # Now run a TCP/IP server and send clients to worker
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(('', port))
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
```

```
            send_handle(worker, client.fileno(), worker_pid)
            client.close()

    if __name__ == '__main__':
        import sys
        if len(sys.argv) != 3:
            print('Usage: server.py server_address port', file=sys.stderr)
            raise SystemExit(1)

        server(sys.argv[1], int(sys.argv[2]))
```

To run this server, you would run a command such as `python3 servermp.py /tmp/servconn 15000`. Here is the corresponding client code:

```
# workermp.py

from multiprocessing.connection import Client
from multiprocessing.reduction import recv_handle
import os
from socket import socket, AF_INET, SOCK_STREAM

def worker(server_address):
    serv = Client(server_address, authkey=b'peekaboo')
    serv.send(os.getpid())
    while True:
        fd = recv_handle(serv)
        print('WORKER: GOT FD', fd)
        with socket(AF_INET, SOCK_STREAM, fileno=fd) as client:
            while True:
                msg = client.recv(1024)
                if not msg:
                    break
                print('WORKER: RECV {!r}'.format(msg))
                client.send(msg)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Usage: worker.py server_address', file=sys.stderr)
        raise SystemExit(1)

    worker(sys.argv[1])
```

To run the worker, you would type `python3 workermp.py /tmp/servconn`. The resulting operation should be exactly the same as the example that used `Pipe()`.

Under the covers, file descriptor passing involves creating a UNIX domain socket and the `sendmsg()` method of sockets. Since this technique is not widely known, here is a different implementation of the server that shows how to pass descriptors using sockets:

```
# server.py
import socket
```

```
import struct

def send_fd(sock, fd):
    '''
    Send a single file descriptor.
    '''
    sock.sendmsg([b'x'],
                 [(socket.SOL_SOCKET, socket.SCM_RIGHTS, struct.pack('i', fd))])
    ack = sock.recv(2)
    assert ack == b'OK'

def server(work_address, port):
    # Wait for the worker to connect
    work_serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    work_serv.bind(work_address)
    work_serv.listen(1)
    worker, addr = work_serv.accept()

    # Now run a TCP/IP server and send clients to worker
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(('',port))
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
        send_fd(worker, client.fileno())
        client.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: server.py server_address port', file=sys.stderr)
        raise SystemExit(1)

    server(sys.argv[1], int(sys.argv[2]))
```

Here is an implementation of the worker using sockets:

```
# worker.py
import socket
import struct

def recv_fd(sock):
    '''
    Receive a single file descriptor
    '''
    msg, ancdata, flags, addr = sock.recvmsg(1,
                                     socket.CMSG_LEN(struct.calcsize('i')))

    cmsg_level, cmsg_type, cmsg_data = ancdata[0]
    assert cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS
    sock.sendall(b'OK')
```

```
        return struct.unpack('i', cmsg_data)[0]

    def worker(server_address):
        serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
        serv.connect(server_address)
        while True:
            fd = recv_fd(serv)
            print('WORKER: GOT FD', fd)
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd) as client:
                while True:
                    msg = client.recv(1024)
                    if not msg:
                        break
                    print('WORKER: RECV {!r}'.format(msg))
                    client.send(msg)

    if __name__ == '__main__':
        import sys
        if len(sys.argv) != 2:
            print('Usage: worker.py server_address', file=sys.stderr)
            raise SystemExit(1)

        worker(sys.argv[1])
```

If you are going to use file-descriptor passing in your program, it is advisable to read more about it in an advanced text, such as *Unix Network Programming* by W. Richard Stevens (Prentice Hall, 1990). Passing file descriptors on Windows uses a different technique than Unix (not shown). For that platform, it is advisable to study the source code to `multiprocessing.reduction` in close detail to see how it works.

11.12. Understanding Event-Driven I/O

Problem

You have heard about packages based on "event-driven" or "asynchronous" I/O, but you're not entirely sure what it means, how it actually works under the covers, or how it might impact your program if you use it.

Solution

At a fundamental level, event-driven I/O is a technique that takes basic I/O operations (e.g., reads and writes) and converts them into events that must be handled by your program. For example, whenever data was received on a socket, it turns into a "receive" event that is handled by some sort of callback method or function that you supply to respond to it. As a possible starting point, an event-driven framework might start with a base class that implements a series of basic event handler methods like this:

```
class EventHandler:
    def fileno(self):
        'Return the associated file descriptor'
        raise NotImplemented('must implement')

    def wants_to_receive(self):
        'Return True if receiving is allowed'
        return False

    def handle_receive(self):
        'Perform the receive operation'
        pass

    def wants_to_send(self):
        'Return True if sending is requested'
        return False

    def handle_send(self):
        'Send outgoing data'
        pass
```

Instances of this class then get plugged into an event loop that looks like this:

```
import select

def event_loop(handlers):
    while True:
        wants_recv = [h for h in handlers if h.wants_to_receive()]
        wants_send = [h for h in handlers if h.wants_to_send()]
        can_recv, can_send, _ = select.select(wants_recv, wants_send, [])
        for h in can_recv:
            h.handle_receive()
        for h in can_send:
            h.handle_send()
```

That's it! The key to the event loop is the `select()` call, which polls file descriptors for activity. Prior to calling `select()`, the event loop simply queries all of the handlers to see which ones want to receive or send. It then supplies the resulting lists to `select()`. As a result, `select()` returns the list of objects that are ready to receive or send. The corresponding `handle_receive()` or `handle_send()` methods are triggered.

To write applications, specific instances of `EventHandler` classes are created. For example, here are two simple handlers that illustrate two UDP-based network services:

```
import socket
import time

class UDPServer(EventHandler):
    def __init__(self, address):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind(address)
```

```
    def fileno(self):
        return self.sock.fileno()

    def wants_to_receive(self):
        return True

class UDPTimeServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(1)
        self.sock.sendto(time.ctime().encode('ascii'), addr)

class UDPEchoServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(8192)
        self.sock.sendto(msg, addr)

if __name__ == '__main__':
    handlers = [ UDPTimeServer(('',14000)), UDPEchoServer(('',15000))  ]
    event_loop(handlers)
```

To test this code, you can try connecting to it from another Python interpreter:

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'',('localhost',14000))
0
>>> s.recvfrom(128)
(b'Tue Sep 18 14:29:23 2012', ('127.0.0.1', 14000))
>>> s.sendto(b'Hello',('localhost',15000))
5
>>> s.recvfrom(128)
(b'Hello', ('127.0.0.1', 15000))
>>>
```

Implementing a TCP server is somewhat more complex, since each client involves the instantiation of a new handler object. Here is an example of a TCP echo client.

```
class TCPServer(EventHandler):
    def __init__(self, address, client_handler, handler_list):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
        self.sock.bind(address)
        self.sock.listen(1)
        self.client_handler = client_handler
        self.handler_list = handler_list

    def fileno(self):
        return self.sock.fileno()

    def wants_to_receive(self):
        return True
```

```
        def handle_receive(self):
            client, addr = self.sock.accept()
            # Add the client to the event loop's handler list
            self.handler_list.append(self.client_handler(client, self.handler_list))

    class TCPClient(EventHandler):
        def __init__(self, sock, handler_list):
            self.sock = sock
            self.handler_list = handler_list
            self.outgoing = bytearray()

        def fileno(self):
            return self.sock.fileno()

        def close(self):
            self.sock.close()
            # Remove myself from the event loop's handler list
            self.handler_list.remove(self)

        def wants_to_send(self):
            return True if self.outgoing else False

        def handle_send(self):
            nsent = self.sock.send(self.outgoing)
            self.outgoing = self.outgoing[nsent:]

    class TCPEchoClient(TCPClient):
        def wants_to_receive(self):
            return True

        def handle_receive(self):
            data = self.sock.recv(8192)
            if not data:
                self.close()
            else:
                self.outgoing.extend(data)

    if __name__ == '__main__':
       handlers = []
       handlers.append(TCPServer(('',16000), TCPEchoClient, handlers))
       event_loop(handlers)
```

The key to the TCP example is the addition and removal of clients from the handler list. On each connection, a new handler is created for the client and added to the list. When the connection is closed, each client must take care to remove themselves from the list.

If you run this program and try connecting with Telnet or some similar tool, you'll see it echoing received data back to you. It should easily handle multiple clients.

Discussion

Virtually all event-driven frameworks operate in a manner that is similar to that shown in the solution. The actual implementation details and overall software architecture might vary greatly, but at the core, there is a polling loop that checks sockets for activity and which performs operations in response.

One potential benefit of event-driven I/O is that it can handle a very large number of simultaneous connections without ever using threads or processes. That is, the `select()` call (or equivalent) can be used to monitor hundreds or thousands of sockets and respond to events occuring on any of them. Events are handled one at a time by the event loop, without the need for any other concurrency primitives.

The downside to event-driven I/O is that there is no true concurrency involved. If any of the event handler methods blocks or performs a long-running calculation, it blocks the progress of everything. There is also the problem of calling out to library functions that aren't written in an event-driven style. There is always the risk that some library call will block, causing the event loop to stall.

Problems with blocking or long-running calculations can be solved by sending the work out to a separate thread or process. However, coordinating threads and processes with an event loop is tricky. Here is an example of code that will do it using the `concurrent.futures` module:

```
from concurrent.futures import ThreadPoolExecutor
import os

class ThreadPoolHandler(EventHandler):
    def __init__(self, nworkers):
        if os.name == 'posix':
            self.signal_done_sock, self.done_sock = socket.socketpair()
        else:
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self.signal_done_sock = socket.socket(socket.AF_INET,
                                                  socket.SOCK_STREAM)
            self.signal_done_sock.connect(server.getsockname())
            self.done_sock, _ = server.accept()
            server.close()

        self.pending = []
        self.pool = ThreadPoolExecutor(nworkers)

    def fileno(self):
        return self.done_sock.fileno()

    # Callback that executes when the thread is done
    def _complete(self, callback, r):
```

```
            self.pending.append((callback, r.result()))
            self.signal_done_sock.send(b'x')

        # Run a function in a thread pool
        def run(self, func, args=(), kwargs={},*,callback):
            r = self.pool.submit(func, *args, **kwargs)
            r.add_done_callback(lambda r: self._complete(callback, r))

        def wants_to_receive(self):
            return True

        # Run callback functions of completed work
        def handle_receive(self):
            # Invoke all pending callback functions
            for callback, result in self.pending:
                callback(result)
                self.done_sock.recv(1)
            self.pending = []
```

In this code, the `run()` method is used to submit work to the pool along with a callback function that should be triggered upon completion. The actual work is then submitted to a `ThreadPoolExecutor` instance. However, a really tricky problem concerns the coordination of the computed result and the event loop. To do this, a pair of sockets are created under the covers and used as a kind of signaling mechanism. When work is completed by the thread pool, it executes the `_complete()` method in the class. This method queues up the pending callback and result before writing a byte of data on one of these sockets. The `fileno()` method is programmed to return the other socket. Thus, when this byte is written, it will signal to the event loop that something has happened. The `handle_receive()` method, when triggered, will then execute all of the callback functions for previously submitted work. Frankly, it's enough to make one's head spin.

Here is a simple server that shows how to use the thread pool to carry out a long-running calculation:

```
    # A really bad Fibonacci implementation
    def fib(n):
        if n < 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

    class UDPFibServer(UDPServer):
        def handle_receive(self):
            msg, addr = self.sock.recvfrom(128)
            n = int(msg)
            pool.run(fib, (n,), callback=lambda r: self.respond(r, addr))

        def respond(self, result, addr):
            self.sock.sendto(str(result).encode('ascii'), addr)
```

```
if __name__ == '__main__':
    pool = ThreadPoolHandler(16)
    handlers = [ pool, UDPFibServer(('',16000))]
    event_loop(handlers)
```

To try this server, simply run it and try some experiments with another Python program:

```
from socket import *
sock = socket(AF_INET, SOCK_DGRAM)
for x in range(40):
    sock.sendto(str(x).encode('ascii'), ('localhost', 16000))
    resp = sock.recvfrom(8192)
    print(resp[0])
```

You should be able to run this program repeatedly from many different windows and have it operate without stalling other programs, even though it gets slower and slower as the numbers get larger.

Having gone through this recipe, should you use its code? Probably not. Instead, you should look for a more fully developed framework that accomplishes the same task. However, if you understand the basic concepts presented here, you'll understand the core techniques used to make such frameworks operate. As an alternative to callback-based programming, event-driven code will sometimes use coroutines. See Recipe 12.12 for an example.

11.13. Sending and Receiving Large Arrays

Problem

You want to send and receive large arrays of contiguous data across a network connection, making as few copies of the data as possible.

Solution

The following functions utilize memoryviews to send and receive large arrays:

```
# zerocopy.py

def send_from(arr, dest):
    view = memoryview(arr).cast('B')
    while len(view):
        nsent = dest.send(view)
        view = view[nsent:]

def recv_into(arr, source):
    view = memoryview(arr).cast('B')
    while len(view):
        nrecv = source.recv_into(view)
        view = view[nrecv:]
```

To test the program, first create a server and client program connected over a socket. In the server:

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.bind(('', 25000))
>>> s.listen(1)
>>> c,a = s.accept()
>>>
```

In the client (in a separate interpreter):

```
>>> from socket import *
>>> c = socket(AF_INET, SOCK_STREAM)
>>> c.connect(('localhost', 25000))
>>>
```

Now, the whole idea of this recipe is that you can blast a huge array through the connection. In this case, arrays might be created by the `array` module or perhaps `numpy`. For example:

```
# Server
>>> import numpy
>>> a = numpy.arange(0.0, 50000000.0)
>>> send_from(a, c)
>>>

# Client
>>> import numpy
>>> a = numpy.zeros(shape=50000000, dtype=float)
>>> a[0:10]
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
>>> recv_into(a, c)
>>> a[0:10]
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>>
```

Discussion

In data-intensive distributed computing and parallel programming applications, it's not uncommon to write programs that need to send/receive large chunks of data. However, to do this, you somehow need to reduce the data down to raw bytes for use with low-level network functions. You may also need to slice the data into chunks, since most network-related functions aren't able to send or receive huge blocks of data entirely all at once.

One approach is to serialize the data in some way—possibly by converting into a byte string. However, this usually ends up making a copy of the data. Even if you do this piecemeal, your code still ends up making a lot of little copies.

This recipe gets around this by playing a sneaky trick with memoryviews. Essentially, a memoryview is an overlay of an existing array. Not only that, memoryviews can be cast to different types to allow interpretation of the data in a different manner. This is the purpose of the following statement:

```
view = memoryview(arr).cast('B')
```

It takes an array `arr` and casts into a memoryview of unsigned bytes.

In this form, the view can be passed to socket-related functions, such as `sock.send()` or `send.recv_into()`. Under the covers, those methods are able to work directly with the memory region. For example, `sock.send()` sends data directly from memory without a copy. `send.recv_into()` uses the memoryview as the input buffer for the receive operation.

The remaining complication is the fact that the socket functions may only work with partial data. In general, it will take many different `send()` and `recv_into()` calls to transmit the entire array. Not to worry. After each operation, the view is sliced by the number of sent or received bytes to produce a new view. The new view is also a memory overlay. Thus, no copies are made.

One issue here is that the receiver has to know in advance how much data will be sent so that it can either preallocate an array or verify that it can receive the data into an existing array. If this is a problem, the sender could always arrange to send the size first, followed by the array data.