Socket Server – Creating network servers

The Socket Server module is a framework for creating network servers. It defines classes for handling synchronous network requests (the server request handler blocks until the request is completed) over TCP, UDP, Unix streams, and Unix datagrams. It also provides mix-in classes for easily converting servers to use a separate thread or process for each request, depending on what is most appropriate for your situation.

Responsibility for processing a request is split between a server class and a request handler class. The server deals with the communication issues (listing on a socket, accepting connections, etc.) and the request handler deals with the "protocol" issues (interpreting incoming data, processing it, sending data back to the client). This division of responsibility means that in many cases you can simply use one of the existing server classes without any modifications, and provide a request handler class for it to work with your protocol.

Server Types

There are five different server classes defined in SocketServer. BaseServer defines the API, and is not really intended to be instantiated and used directly. TCPServer uses TCP/IP sockets to communicate. UDPServer uses datagram sockets. UnixStreamServer andUnixDatagramServer use Unix-domain sockets and are only available on Unix platforms.

Server Objects

To construct a server, pass it an address on which to listen for requests and a request handler class (not instance). The address format depends on the server type and the socket family used. Refer to the socket module documentation for details.

Once the server object is instantiated, use either handle_request() or serve_forever() to process requests. The serve_forever() method simply calls handle_request() in an infinite loop, so if you need to integrate the server with another event loop or use select() to monitor several sockets for different servers, you could call handle_request() on your own. See the example below for more detail.

Implementing a Server

If you are creating a server, it is usually possible to re-use one of the existing classes and simply provide a custom request handler class. If that does not meet your needs, there are several methods of BaseServer available to override in a subclass:

- verify_request(request, client_address) - Return True to process the request or False to ignore it. You could, for example, refuse requests from an IP range if you want to block certain clients from accessing the server.

- process_request(request, client_address) - Typically just calls finish_request() to actually do the work. It can also create a separate thread or process, as the mix-in classes do (see below).

- finish_request(request, client_address) - Creates a request handler instance using the class given to the server's constructor. Calls handle() on the request handler to process the request.

Request Handlers

Request handlers do most of the work of receiving incoming requests and deciding what action to take. The handler is responsible for implementing the "protocol" on top of the socket layer (for example, HTTP or XML-RPC). The request handler reads the request from the incoming data channel, processes it, and writes a response back out. There are 3 methods available to be over-ridden.

- setup() - Prepare the request handler for the request. In the StreamRequestHandler, for example, the setup() method creates file-like objects for reading from and writing to the socket.

- handle() - Do the real work for the request. Parse the incoming request, process the data, and send a response.

- finish() - Clean up anything created during setup().

In many cases, you can simply provide a handle() method.

Echo Example

Let’s look at a simple server/request handler pair that accepts TCP connectcions and echos back any data sent by the client. The only method that actually needs to be provided in the sample code is EchoRequestHandler.handle(), but all of the methods described above are overridden to insert logging calls so the output of the sample program illustrates the sequence of calls made.

The only thing left is to have simple program that creates the server, runs it in a thread, and connects to it to illustrate which methods are called as the data is echoed back.

```
import logging

import sys

import SocketServer


logging.basicConfig(level=logging.DEBUG,

                    format='%(name)s: %(message)s',

                    )


class EchoRequestHandler(SocketServer.BaseRequestHandler):


    def __init__(self, request, client_address, server):

        self.logger = logging.getLogger('EchoRequestHandler')

        self.logger.debug('__init__')

        SocketServer.BaseRequestHandler.__init__(self, request, client_address, server)

        return
```

```
    def setup(self):
        self.logger.debug('setup')
        return SocketServer.BaseRequestHandler.setup(self)

    def handle(self):
        self.logger.debug('handle')

        # Echo the back to the client
        data = self.request.recv(1024)
        self.logger.debug('recv()->"%s"', data)
        self.request.send(data)
        return

    def finish(self):
        self.logger.debug('finish')
        return SocketServer.BaseRequestHandler.finish(self)

class EchoServer(SocketServer.TCPServer):

    def __init__(self, server_address, handler_class=EchoRequestHandler):
        self.logger = logging.getLogger('EchoServer')
        self.logger.debug('__init__')
```

```
    SocketServer.TCPServer.__init__(self, server_address, handler_class)
    return

def server_activate(self):
    self.logger.debug('server_activate')
    SocketServer.TCPServer.server_activate(self)
    return

def serve_forever(self):
    self.logger.debug('waiting for request')
    self.logger.info('Handling requests, press <Ctrl-C> to quit')
    while True:
        self.handle_request()
    return

def handle_request(self):
    self.logger.debug('handle_request')
    return SocketServer.TCPServer.handle_request(self)

def verify_request(self, request, client_address):
    self.logger.debug('verify_request(%s, %s)', request, client_address)
    return SocketServer.TCPServer.verify_request(self, request, client_address)
```

```
    def process_request(self, request, client_address):
        self.logger.debug('process_request(%s, %s)', request, client_address)
        return SocketServer.TCPServer.process_request(self, request, client_address)

    def server_close(self):
        self.logger.debug('server_close')
        return SocketServer.TCPServer.server_close(self)

    def finish_request(self, request, client_address):
        self.logger.debug('finish_request(%s, %s)', request, client_address)
        return SocketServer.TCPServer.finish_request(self, request, client_address)

    def close_request(self, request_address):
        self.logger.debug('close_request(%s)', request_address)
        return SocketServer.TCPServer.close_request(self, request_address)

if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0) # let the kernel give us a port
    server = EchoServer(address, EchoRequestHandler)
    ip, port = server.server_address # find out what port we were given
```

```
t = threading.Thread(target=server.serve_forever)
t.setDaemon(True) # don't hang on exit
t.start()

logger = logging.getLogger('client')
logger.info('Server on %s:%s', ip, port)

# Connect to the server
logger.debug('creating socket')
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
logger.debug('connecting to server')
s.connect((ip, port))

# Send the data
message = 'Hello, world'
logger.debug('sending data: "%s"', message)
len_sent = s.send(message)

# Receive a response
logger.debug('waiting for response')
response = s.recv(len_sent)
logger.debug('response from server: "%s"', response)
```

```
    # Clean up
    logger.debug('closing socket')
    s.close()
    logger.debug('done')
    server.socket.close()
```

The output for the program should look something like this:

```
$ python SocketServer_echo.py

EchoServer: __init__
EchoServer: server_activate
EchoServer: waiting for request
EchoServer: Handling requests, press <Ctrl-C> to quit
client: Server on 127.0.0.1:54406
EchoServer: handle_request
client: creating socket
client: connecting to server
client: sending data: "Hello, world"
EchoServer: verify_request(<socket._socketobject object at 0x1004f08a0>, ('127.0.0.1', 54407))
EchoServer: process_request(<socket._socketobject object at 0x1004f08a0>, ('127.0.0.1', 54407))
client: waiting for response
```

EchoServer: finish_request(<socket._socketobject object at 0x1004f08a0>, ('127.0.0.1', 54407))

EchoRequestHandler: __init__

EchoRequestHandler: setup

EchoRequestHandler: handle

EchoRequestHandler: recv()->"Hello, world"

EchoRequestHandler: finish

EchoServer: close_request(<socket._socketobject object at 0x1004f08a0>)

EchoServer: handle_request

client: response from server: "Hello, world"

client: closing socket

client: done

The port number used will change each time you run it, as the kernel allocates an available port automatically. If you want the server to listen on a specific port each time you run it, provide that number in the address tuple instead of the 0.

Here is a simpler version of the same thing, without the logging:

```
import SocketServer


class EchoRequestHandler(SocketServer.BaseRequestHandler):


    def handle(self):
        # Echo the back to the client
        data = self.request.recv(1024)
        self.request.send(data)
```

```
        return

if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0) # let the kernel give us a port
    server = SocketServer.TCPServer(address, EchoRequestHandler)
    ip, port = server.server_address # find out what port we were given

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    message = 'Hello, world'
    print 'Sending : "%s"' % message
    len_sent = s.send(message)
```

```
    # Receive a response
    response = s.recv(len_sent)
    print 'Received: "%s"' % response

    # Clean up
    s.close()
    server.socket.close()
```

In this case, no special server class is required since the TCPServer handles all of the server requirements.

```
$ python SocketServer_echo_simple.py

Sending : "Hello, world"
Received: "Hello, world"
```

Threading and Forking

Adding threading or forking support to a server is as simple as including the appropriate mix-in in the class hierarchy for the server. The mix-in classes override process_request() to start a new thread or process when a request is ready to be handled, and the work is done in the new child.

For threads, use the ThreadingMixIn:

```
import threading
import SocketServer

class ThreadedEchoRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
```

```
        # Echo the back to the client
        data = self.request.recv(1024)
        cur_thread = threading.currentThread()
        response = '%s: %s' % (cur_thread.getName(), data)
        self.request.send(response)
        return

class ThreadedEchoServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass

if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0) # let the kernel give us a port
    server = ThreadedEchoServer(address, ThreadedEchoRequestHandler)
    ip, port = server.server_address # find out what port we were given

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()
    print 'Server loop running in thread:', t.getName()
```

```
    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    message = 'Hello, world'
    print 'Sending : "%s"' % message
    len_sent = s.send(message)

    # Receive a response
    response = s.recv(1024)
    print 'Received: "%s"' % response

    # Clean up
    s.close()
    server.socket.close()
```

The response from the server includes the id of the thread where the request is handled:

```
$ python SocketServer_threaded.py

Server loop running in thread: Thread-1
Sending : "Hello, world"
Received: "Thread-2: Hello, world"
```

To use separate processes, use the ForkingMixIn:

```
import os
import SocketServer

class ForkingEchoRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        # Echo the back to the client
        data = self.request.recv(1024)
        cur_pid = os.getpid()
        response = '%s: %s' % (cur_pid, data)
        self.request.send(response)
        return

class ForkingEchoServer(SocketServer.ForkingMixIn, SocketServer.TCPServer):
    pass

if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0) # let the kernel give us a port
    server = ForkingEchoServer(address, ForkingEchoRequestHandler)
    ip, port = server.server_address # find out what port we were given
```

```
    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()
    print 'Server loop running in process:', os.getpid()

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    message = 'Hello, world'
    print 'Sending : "%s"' % message
    len_sent = s.send(message)

    # Receive a response
    response = s.recv(1024)
    print 'Received: "%s"' % response

    # Clean up
    s.close()
    server.socket.close()
```

In this case, the process id of the child is included in the response from the server:

$ python SocketServer_forking.py

Server loop running in process: 14659

Sending : "Hello, world"

Received: "14660: Hello, world"

Exception in thread Thread-1 (most likely raised during interpreter shutdown):

Source: http://bip.weizmann.ac.il/course/python/PyMOTW/PyMOTW/docs/SocketServer/index.html#module-SocketServer