Chapter 39
Sockets in Python

39.1 Introduction

A *Socket* is an end point in a communication link between separate processes. In Python sockets are objects which provide a way of exchanging information between two processes in a straight forward and platform independent manner.

In this chapter we will introduce the basic idea of socket communications and then presents a simple socket server and client application.

39.2 Socket to Socket Communication

When two operating system level processes wish to communicate, they can do so via *sockets*. Each process has a socket which is connected to the others socket. One process can then write information out to the socket, while the second process can read information in from the socket.

Associated with each socket are two streams, one for input and one for output. Thus, to pass information from one process to another, you write that information out to the output stream of one socket object and read it from the input stream of another socket object (assuming the two sockets are connected).

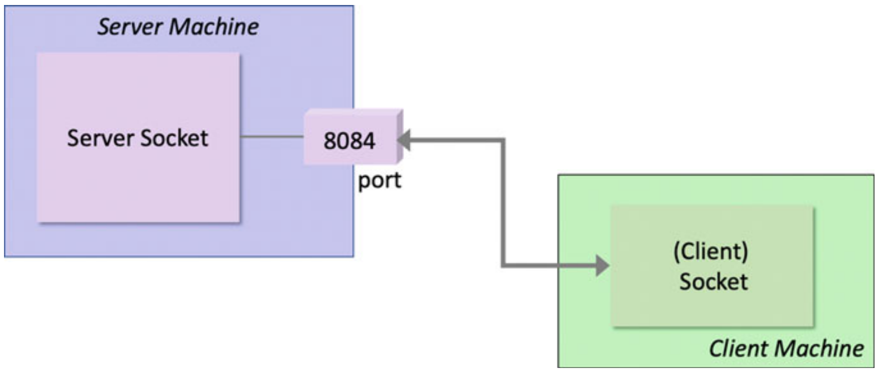
Several different types of sockets are available, however in this chapter we will focus on TCP/IP sockets. Such a socket is a connection-oriented socket that will provide a guarantee of delivery of data (or notification of the failure to deliver the data). TCP/IP, or the Transmission Control Protocol/Internet Protocol, is a suite of communication protocols used to interconnect network devices on the internet or in a private intranet. TCP/IP actually specifies how data is exchanged between programs over the internet by providing end-to-end communications that identify how the data should be broken down into packets, addressed, transmitted, routed and received at the destination.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_39

39.3 Setting Up a Connection

To set up the connection, one process must be running a program that is waiting for a connection while the other must try to connect up to the first program. The first is referred to as a server socket while the second just as a socket.

For the second process to connect to the first (the server socket) it must know what machine the first is running on and which port it is connected to.



For example, in the above diagram the server socket connects to port 8084. In turn the client socket connects to the machine on which the server is executing and to port number 8084 on that machine.

Nothing happens until the server socket accepts the connection. At that point the sockets are connected, and the socket streams are bound to each other. This means that the server's output stream is connected to the Client socket input stream and vice versa.

39.4 An Example Client Server Application

39.4.1 The System Structure

The above diagram illustrates the basic structure of the system we are trying to build. There will be a server object running on one machine and a client object running on another. The client will connect up to the server using sockets in order to obtain information.

The actual application being implemented in this example, is an address book look up application. The addresses of employees of a company are held in a dictionary. This dictionary is set up in the server program but could equally be held in a database etc. When a client connects up to the server it can obtain an employees' office address.

39.4.2 Implementing the Server Application

We shall describe the server application first. This is the Python application program that will service requests from client applications. To do this it must provide a server socket for clients to connect to. This is done by first binding a server socket to a port on the server machine. The server program must then listen for incoming connections.

The listing presents the source code for the `Server` program.

```
import socket

def main():
    # Setup names and offices
    addresses = {'JOHN': 'C45',
                 'DENISE': 'C44',
                 'PHOEBE': 'D52',
                 'ADAM': 'B23'}

    print('Starting Server')
    print('Create the socket')
    sock = socket.socket(socket.AF_INET,
                         socket.SOCK_STREAM)

    print('Bind the socket to the port')
    server_address = (socket.gethostname(),
                      8084)
    print('Starting up on', server_address)
    sock.bind(server_address)

    # specifies the number of connections allowed
    print('Listen for incoming connections')
    sock.listen(1)
    while True:
        print('Waiting for a connection')
        connection, client_address =
                            sock.accept()
        try:
            print('Connection from',
                  client_address)
            while True:
                data =
                 connection.recv(1024).decode()
                print('Received: ', data)
                if data:
                    key = str(data).upper()
                    response = addresses[key]
                    print('sending data back
                              to the client: ',
                          response)
                    connection.sendall(
                             response.encode())
                else:
                    print('No more data from',
                          client_address)
                    break
        finally:
            connection.close()

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

The Server in the above listing sets up the `addresses` to contain a Dictionary of the names and addresses.

It then waits for a client to connect to it. This is done by creating a socket and binding it to a specific port (in this case port 8084) using:

```
print('Create the socket')
sock = socket.socket(socket.AF_INET,
                      socket.SOCK_STREAM)
print('Bind the socket to the port')
server_address = (socket.gethostname(),
                  8084)
```

The construction of the socket object is discussed in more detail in the next section.

Next the server listens for a connection from a client. Note that the `sock.listen()` method takes the value 1 indicating that it will handle one connection at a time.

An infinite loop is then set up to run the server. When a connection is made from a client, both the connection and the client address are made available. While there is data available from the client, it is read using the recv function. Note that the data received from the client is assumed to be a string. This is then used as a key to look the address up in the address Dictionary.

Once the address is obtained it can be sent back to the client. In Python 3 it is necessary to `decode()` and `encoded()` the string format to the raw data transmitted via the socket streams.

Note you should always *close* a socket when you have finished with it.

39.5 Socket Types and Domains

When we created the socket class above, we passed in two arguments to the socket constructor:

```
socket(socket.AF_INET, socket.SOCK_STREAM)
```

To understand the two values passed into the `socket()` constructor it is necessary to understand that Sockets are characterised according to two properties; their *domain* and their *type*.

The *domain* of a socket essentially defines the communications protocols that are used to transfer the data from one process to another. It also incorporates how sockets are named (so that they can be referred to when establishing the communication).

Two *standard* domains are available on Unix systems; these are `AF_UNIX` which represents intra-system communications, where data is moved from process to process through kernel memory buffers. `AF_INET` represents communication using the TCP/IP protocol suite; in which processes may be on the same machine or on different machines.

- A socket's *type* indicates how the data is transferred through the socket. There are essentially two options here:
- Datagram which sockets support a *message-based* model where no connection is involved, and communication is not guaranteed to be reliable.
- Stream sockets that support a *virtual circuit* model, where data is exchanged as a byte stream and the connection is reliable.

Depending on the domain, further socket types may be available, such as those that support message passing on a reliable connection.

39.6 Implementing the Client Application

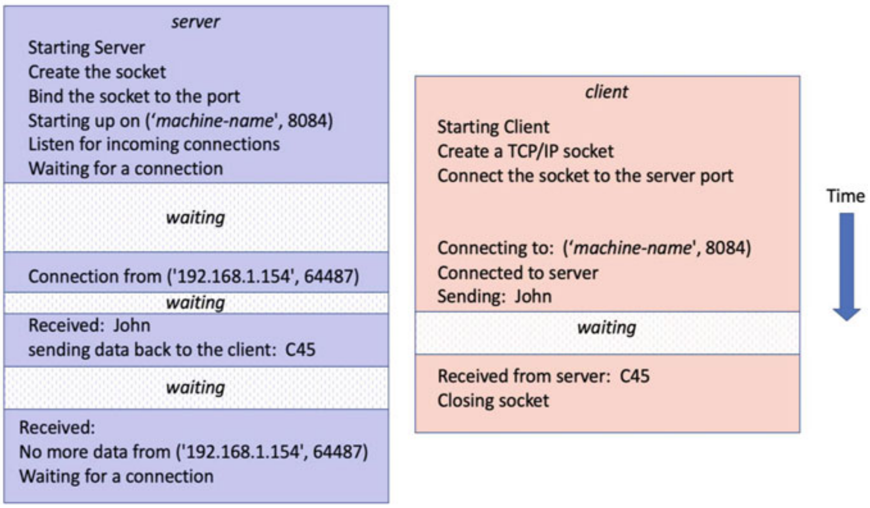
The client application is essentially a very simple program that creates a link to the server application. To do this it creates a socket object that connects to the servers' host machine, and in our case this socket is connected to port 8084.

Once a connection has been made the client can then send the encoded message string to the server. The server will then send back a response which the client must decode. It then closes the connection.

The implementation of the client is given below:

```
import socket

def main():
    print('Starting Client')
    print('Create a TCP/IP socket')
    sock = socket.socket(socket.AF_INET,
                         socket.SOCK_STREAM)
    print('Connect the socket to the server port')
    server_address = (socket.gethostname(),
                      8084)
    print('Connecting to: ', server_address)
    sock.connect(server_address)
    print('Connected to server')
    try:
        print('Send data')
        message = 'John'
        print('Sending: ', message)
        sock.send(message.encode())
        data = sock.recv(1024).decode()
        print('Received from server: ', data)
    finally:
        print('Closing socket')
        sock.close()

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

The output from the two programs needs to be considered together.



```
Starting Server
Create the socket
Bind the socket to the port
Starting up on ('machine-name', 8084)
Listen for incoming connections
Waiting for a connection
Connection from ('192.168.1.154', 64487)
Received:  John
sending data back to the client:  C45
Received:
No more data from ('192.168.1.154', 64487)
Waiting for a connection
```

```
Starting Client
Create a TCP/IP socket
Connect the socket to the server port
Connecting to:  ('machine-name', 8084)
Connected to server
Sending:  John
Received from server:  C45
Closing socket
```

As you can see from this diagram, the server waits for a connection from the client. When the client connects to the server; the server waits to receive data from the client. At this point the client must wait for data to be sent to it from the server. The server then sets up the response data and sends it back to the client. The client receives this and prints it out and closes the connection. In the meantime, the server has been waiting to see if there is any more data from the client; as the client closes the connection the server knows that the client has finished and returns to waiting for the next connection.

39.7 The Socketserver Module

In the above example, the server code is more complex than the client; and this is for a single threaded server; life can become much more complicated if the server is expected to be a multi-threaded server (that is a server that can handle multiple requests from different clients at the same time).

However, the `serversocket` module provides a more convenient, object-oriented approach to creating a server. Much of the *boiler plate* code needed in such applications is defined in classes, with the developer only having to provide their own classes or override methods to define the specific functionality required.

There are five different server classes defined in the `socketserver` module.

- `BaseServer` is the root of the Server class hierarchy; it is not really intended to be instantiated and used directly. Instead it is extended by `TCPServer` and other classes.
- `TCPServer` uses TCP/IP sockets to communicate and is probably the most commonly used type of socket server.
- `UDPServer` provides access to datagram sockets.
- `UnixStreamServer` and `UnixDatagramServer` use Unix-domain sockets and are only available on Unix platforms.

Responsibility for processing a request is split between a *server* class and a *request handler* class. The server deals with the *communication* issues (listening on a socket and port, accepting connections, etc.) and the request handler deals with the *request* 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 custom request handler class for it to work with.

The following example defines a request handler that is plugged into the `TCPServer` when it is constructed. The request handler defines a method `handle()` that will be expected to handle the request processing.

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The RequestHandler class for the server.
    """

    def __init__(self, request, client_address, server):
        print('Setup names and offices')
        self.addresses = {'JOHN': 'C45',
                          'DENISE': 'C44',
                          'PHOEBE': 'D52',
                          'ADAM': 'B23'}
        super().__init__(request, client_address, server)

    def handle(self):
        print('In Handle')
        # self.request is the TCP socket connected
        # to the client
        data = self.request.recv(1024).decode()
        print('data received:', data)
        key = str(data).upper()
        response = self.addresses[key]
        print('response:', response)
        # Send the result back to the client
        self.request.sendall(response.encode())

def main():
    print('Starting server')
    server_address = ('localhost', 8084)
    print('Creating server')
    server =
       socketserver.TCPServer(server_address,
                                   MyTCPHandler)
    print('Activating server')
    server.serve_forever()

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

Note that the previous client application does not need to change at all; the server changes are hidden from the client.

However, this is still a single threaded server. We can very simply make it into a multi-threaded server (one that can deal with multiple requests concurrently) by mixing the `socketserver.ThreadingMixIn` into the `TCPServer`. This can be done by defining a new class that is nothing more than a class that extends both

ThreadingMixIn and TCPServer and creating an instane of this new class instead of the TCPServer directly. For example:

```
class ThreadedEchoServer(
            socketserver.ThreadingMixIn,
            socketserver.TCPServer):
    pass

def main():
    print('Starting')
    address = ('localhost', 8084)
    server = ThreadedEchoServer(address,
                                  MyTCPHandler)
    print('Activating server')
    server.serve_forever()
```

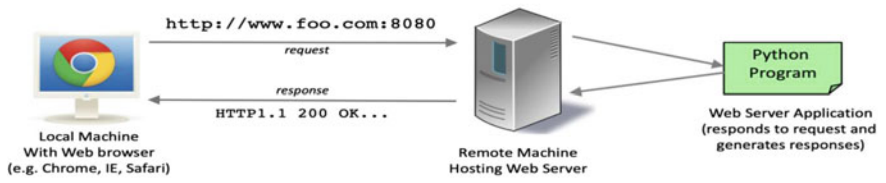
In fact you do not even need to create your own class (such as the ThreadedEchoServer) as the socketserver.ThreadingTCPServer has been provided as a default mixing of the TCPServer and the ThreadingMixIn classes. We could therefore just write:

```
def main():
    print('Starting')
    address = ('localhost', 8084)
    server = socketserver.ThreadedEchoServer(address,
                                  MyTCPHandler)
    print('Activating server')
    server.serve_forever()
```

39.8 HTTP Server

In addition to the TCPServer you also have available a http.server.HTTPServer; this can be used in a similar manner to the TCPServer, but is used to create servers that respond to the HTTP protocol used by web browsers. In other words it can be used to create a very simple Web Server (although it should be noted that it is really only suitable for creating test web servers as it only implements very basic security checks).

It is probably worth a short aside to illustrate how a web server and a web browser interact. The following diagram illustrates the basic interactions:



In the above diagram the user is using a browser (such as Chrome, IE or Safari) to access a web server. The browser is running on their local machine (which could be a PC, a Mac, a Linux box, an iPad, a Smart Phone etc.).

To access the web server they enter a URL (Universal Resource Locator) address into their browser. In the example this is the URL www.foo.com. It also indicates that they want to connect up to port 8080 (rather than the default port 80 used for HTTP connections). The remote machine (which is the one indicated by the address www.foo.com) receives this request and determines what to do with it. If there is no program monitoring port 8080 it will reject the request. In our case we have a Python Program (which is actually the web server program) listening to that port and it is passed the request. It will then handle this request and generate a response message which will be sent back to the browser on the users local machine. The response will indicate which version of the HTTP protocol it supports, whether everything went ok or not (this is the 200 code in the above diagram - you may have seen the code 404 indicating that a web page was not found etc.). The browser on the local machine then renders the data as a web page or handles the data as appropriate etc.

To create a simple Python web server the `http.server.HTTPServer` can be used directly or can be subclassed along with the `socketserver.ThreadingMixIn` to create a multi-threaded web server, for example:

```
class ThreadingHTTPServer(ThreadingMixIn, HTTPServer):
    """Simple multi-threaded HTTP server """
    pass
```

Since Python 3.7 the `http.server` module now provides exactly this class as a built in facility and it is thus no longer necessary to define it yourself (see `http.server.ThreadingHTTPServer`).

To handle HTTP requests you must implement one of the HTTP request methods such as `do_GET()`, or `do_POST()`. Each of these maps to a type of HTTP request, for example:

- `do_GET()` maps to a HTTP Get request that is generated if you type a web address into the URL bar of a web browser or
- `do_POST()` maps to a HTTP Post request that is used for example, when a form on a web page is used to submit data to a web server.

The `do_GET(self)` or `do_POST(self)` method must then handle any input supplied with the request and generate any appropriate responses back to the browser. This means that it must follow the HTTP protocol.

The following short program creates a simple web server that will generate a welcome message and the current time as a response to a GET request. It does this by using the `datetime` module to create a time stamp of the date and time using the `today()` function. This is converted into a byte array using the UTF-8 character encoding (UTF-8 is the most widely used way to represent text within web pages). We need a byte array as that is what will be executed by the `write()` method later on.

Having done this there are various items of meta data that need to be set up so that the browser knows what data it is about to receive. This meta data is known as *header* data and can including the type of content being sent and the amount of data (content) being transmitted. In our very simple case we need to tell it that we are sending it plain text (rather than the HTML used to describe a typical web page) via the 'Content-type' header information. We also need to tell it how much data we are sending using the content length. We can then indicate that we have finished defining the header information and are now sending the actual data.

The data itself is sent via the `wfile` attribute inherited from the `BaseHTTPRequestHandler`. There are infact two related attributes `rfile` and `wfile`:

- `rfile` this is an input stream that allows you to read input data (which is not being used in this example).
- `wfile` holds the output stream that can be used to write (send) data to the browser. This object provides a method `write()` that takes a byte-like object that is written out to (eventually) the browser.

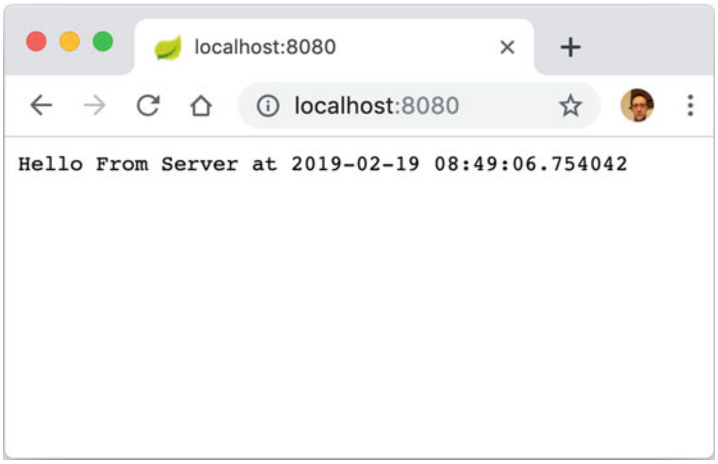
A `main()` method is used to set up the HTTP server which follows the pattern used for the `TCPServer`; however the client of this server will be a web browser.

```
from http.server import BaseHTTPRequestHandler,
ThreadingHTTPServer
from datetime import datetime

class MyHttpRequestHandler(BaseHTTPRequestHandler):
    """Very simple request handler. Only supports GET."""

    def do_GET(self):
        print("do_GET() starting to process request")
        welcome_msg = 'Hello From Server at ' +
str(datetime.today())
        byte_msg = bytes(welcome_msg, 'utf-8')
        self.send_response(200)
        self.send_header("Content-type", 'text/plain; charset-
utf-8')
```

```
        self.send_header('Content-length', str(len(byte_msg)))
        self.end_headers()
        print('do_GET() replying with message')
        self.wfile.write(byte_msg)
def main():
    print('Setting up server')
    server_address = ('localhost', 8080)
    httpd = ThreadingHTTPServer(server_address,
MyHttpRequestHandler)
    print('Activating HTTP server')
    httpd.serve_forever()

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

Once the server is up and running, it is possible to connect to the server using a browser and by entering an appropriate web address into the browsers' URL field. This means that in your browser (assuming it is running on the same machine as the above program) you only need to type into the URL bar `http://localhost:8080` (this indicates you want to use the http protocol to connect up to the local machine at port 8080).

When you do this you should see the welcome message with the current date and time:



```
Hello From Server at 2019-02-19 08:49:06.754042
```

39.9 Online Resources

See the following online resources for information on the topics in this chapter:

- https://docs.python.org/3/howto/sockets.html tutorial on programming sockets in Python.
- https://pymotw.com/3/socket/tcp.html the Python Module of the Week TCP page.
- https://pymotw.com/3/socketserver/index.html The Python Module of the Week page on SocketServer.
- https://docs.python.org/3/library/http.server.html HTTP Servers Python documentation.
- https://pymotw.com/3/http.server/index.html The Python Module of the Week page on the `http.server` module.
- https://www.redbooks.ibm.com/pubs/pdfs/redbooks/gg243376.pdf a PDF tutorial book from IBM on TCP/IP.
- http://flask.pocoo.org/ for more information the Flask micro framework for web development.
- https://www.djangoproject.com/ provides information on the Django framework for creating web applications.

39.10 Exercises

The aim of this exercise is to explore working with TCP/IP sockets.

You should create a TCP server that will receive a string from a client.

A check should then be made to see what information the string indicates is required, supported inputs are:

- ‘`Date`’ which should result in the current date being returned.
- ‘`Time`’ which should result in the current time being returned.
- ‘`Date and Time`’ which should result in the current date and time being returned.
- Anything else should result in the input string being returned to the client in upper case with the message ‘UNKNOWN OPTION’: preceding the string.

The result is then sent back to the client.

You should then create a client program to call the server. The client program can request input from the user in the form of a string and submit that string to the server. The result returned by the server should be displayed in the client before prompting the user for the next input. If the user enters -1 as input then the program should terminate.

An example of the type of output the client might generate is given below to illustrate the general aim of the exercise:

```
Starting Client
Please provide an input (Date, Time, DataAndTime or -1 to
exit): Date
Connected to server
Sending data
Received from server:  2019-02-19
Closing socket
Please provide an input (Date, Time, DataAndTime or -1 to
exit): Time
Connected to server
Sending data
Received from server:  15:50:40
Closing socket
Please provide an input (Date, Time, DataAndTime or -1 to
exit): DateAndTime
Connected to server
Sending data
Received from server:  2019-02-19 15:50:44.720747
Closing socket
Please provide an input (Date, Time, DataAndTime or -1 to
exit): -1
```