TCP/IP Client and Server

Sockets can be configured to act as a *server* and listen for incoming messages, or connect to other applications as a *client*. After both ends of a TCP/IP socket are connected, communication is bi-directional.

Echo Server

This sample program, based on the one in the standard library documentation, receives incoming messages and echos them back to the sender. It starts by creating a TCP/IP socket, then `bind()` is used to associate the socket with the server address. In this case, the address is `localhost`, referring to the current server, and the port number is 10000.

```
# socket_echo_server.py

import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the port
server_address = ('localhost', 10000)
print('starting up on {} port {}'.format(*server_address))
sock.bind(server_address)

# Listen for incoming connections
sock.listen(1)

while True:
    # Wait for a connection
    print('waiting for a connection')
    connection, client_address = sock.accept()
    try:
        print('connection from', client_address)

        # Receive the data in small chunks and retransmit it
        while True:
            data = connection.recv(16)
            print('received {!r}'.format(data))
            if data:
                print('sending data back to the client')
                connection.sendall(data)
            else:
                print('no data from', client_address)
                break

    finally:
        # Clean up the connection
        connection.close()
```

Calling `listen()` puts the socket into server mode, and `accept()` waits for an incoming connection. The integer argument is the number of connections the system should queue up in the background before rejecting new clients. This example only expects to work with one connection at a time.

`accept()` returns an open connection between the server and client, along with the address of the client. The connection is actually a different socket on another port (assigned by the kernel). Data is read from the connection with `recv()` and transmitted with `sendall()`.

When communication with a client is finished, the connection needs to be cleaned up using `close()`. This example uses a `try:finally` block to ensure that `close()` is always called, even in the event of an error.

Echo Client

The client program sets up its `socket` differently from the way a server does. Instead of binding to a port and listening, it uses `connect()` to attach the socket directly to the remote address.

```
# socket_echo_client.py

import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect the socket to the port where the server is listening
server_address = ('localhost', 10000)
print('connecting to {} port {}'.format(*server_address))
sock.connect(server_address)

try:

    # Send data
    message = b'This is the message.  It will be repeated.'
    print('sending {!r}'.format(message))
    sock.sendall(message)

    # Look for the response
    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print('received {!r}'.format(data))

finally:
    print('closing socket')
    sock.close()
```

After the connection is established, data can be sent through the `socket` with `sendall()` and received with `recv()`, just as in the server. When the entire message is sent and a copy received, the socket is closed to free up the port.

Client and Server Together

The client and server should be run in separate terminal windows, so they can communicate with each other. The server output shows the incoming connection and data, as well as the response sent back to the client.

```
$ python3 socket_echo_server.py
starting up on localhost port 10000
waiting for a connection
connection from ('127.0.0.1', 65141)
received b'This is the mess'
sending data back to the client
received b'age.  It will be'
sending data back to the client
received b' repeated.'
sending data back to the client
received b''
no data from ('127.0.0.1', 65141)
waiting for a connection
```

The client output shows the outgoing message and the response from the server.

```
$ python3 socket_echo_client.py
connecting to localhost port 10000
sending b'This is the message.  It will be repeated.'
received b'This is the mess'
received b'age.  It will be'
received b' repeated.'
closing socket
```

Easy Client Connections

Easy Client Connections

TCP/IP clients can save a few steps by using the convenience function `create_connection()` to connect to a server. The function takes one argument, a two-value tuple containing the address of the server, and derives the best address to use for the connection.

```
# socket_echo_client_easy.py
```

```
import socket
import sys


def get_constants(prefix):
    """Create a dictionary mapping socket module
    constants to their names.
    """
    return {
        getattr(socket, n): n
        for n in dir(socket)
        if n.startswith(prefix)
    }


families = get_constants('AF_')
types = get_constants('SOCK_')
protocols = get_constants('IPPROTO_')

# Create a TCP/IP socket
sock = socket.create_connection(('localhost', 10000))

print('Family  :', families[sock.family])
print('Type    :', types[sock.type])
print('Protocol:', protocols[sock.proto])
print()

try:

    # Send data
    message = b'This is the message.  It will be repeated.'
    print('sending {!r}'.format(message))
    sock.sendall(message)

    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print('received {!r}'.format(data))

finally:
    print('closing socket')
    sock.close()
```

`create_connection()` uses `getaddrinfo()` to find candidate connection parameters, and returns a `socket` opened with the first configuration that creates a successful connection. The `family`, `type`, and `proto` attributes can be examined to determine the type of `socket` being returned.

```
$ python3 socket_echo_client_easy.py
Family  : AF_INET
Type    : SOCK_STREAM
Protocol: IPPROTO_TCP

sending b'This is the message.  It will be repeated.'
received b'This is the mess'
received b'age.  It will be'
received b' repeated.'
closing socket
```

Choosing an Address for Listening

Choosing an Address for Listening

It is important to bind a server to the correct address, so that clients can communicate with it. The previous examples all used `'localhost'` as the IP address, which limits connections to clients running on the same server. Use a public address of the server, such as the value returned by `gethostname()`, to allow other hosts to connect. This example modifies the echo server to listen on an address specified via a command line argument.

```
# socket_echo_server_explicit.py

import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the address given on the command line
server_name = sys.argv[1]
server_address = (server_name, 10000)
print('starting up on {} port {}'.format(*server_address))
sock.bind(server_address)
sock.listen(1)

while True:
    print('waiting for a connection')
    connection, client_address = sock.accept()
    try:
        print('client connected:', client_address)
        while True:
            data = connection.recv(16)
            print('received {!r}'.format(data))
            if data:
                connection.sendall(data)
            else:
                break
    finally:
        connection.close()
```

A similar modification to the client program is needed before the server can be tested.

```
# socket_echo_client_explicit.py

import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect the socket to the port on the server
# given by the caller
server_address = (sys.argv[1], 10000)
print('connecting to {} port {}'.format(*server_address))
sock.connect(server_address)

try:

    message = b'This is the message.  It will be repeated.'
    print('sending {!r}'.format(message))
    sock.sendall(message)

    amount_received = 0
    amount_expected = len(message)
    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print('received {!r}'.format(data))

finally:
    sock.close()
```

After starting the server with the argument `hubert`, the `netstat` command shows it listening on the address for the named host.

host.

```
$ host hubert.hellfly.net

hubert.hellfly.net has address 10.9.0.6

$ netstat -an | grep 10000

Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
...
tcp4       0      0  10.9.0.6.10000         *.*                    LISTEN
...
```

Running the client on another host, passing `hubert.hellfly.net` as the host where the server is running, produces:

```
$ hostname

apu

$ python3 ./socket_echo_client_explicit.py hubert.hellfly.net
connecting to hubert.hellfly.net port 10000
sending b'This is the message.  It will be repeated.'
received b'This is the mess'
received b'age.  It will be'
received b' repeated.'
```

And the server output is:

```
$ python3 socket_echo_server_explicit.py hubert.hellfly.net
starting up on hubert.hellfly.net port 10000
waiting for a connection
client connected: ('10.9.0.10', 33139)
received b''
waiting for a connection
client connected: ('10.9.0.10', 33140)
received b'This is the mess'
received b'age.  It will be'
received b' repeated.'
received b''
waiting for a connection
```

Many servers have more than one network interface, and therefore more than one IP address. Rather than running separate copies of a service bound to each IP address, use the special address `INADDR_ANY` to listen on all addresses at the same time. Although `socket` defines a constant for `INADDR_ANY`, it is an integer value and must be converted to a dotted-notation string address before it can be passed to `bind()`. As a shortcut, use "`0.0.0.0`" or an empty string (`''`) instead of doing the conversion.

```
# socket_echo_server_any.py

import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the address given on the command line
server_address = ('', 10000)
sock.bind(server_address)
print('starting up on {} port {}'.format(*sock.getsockname()))
sock.listen(1)

while True:
    print('waiting for a connection')
    connection, client_address = sock.accept()
    try:
        print('client connected:', client_address)
        while True:
            data = connection.recv(16)
            print('received {!r}'.format(data))
```

```
            if data:
                connection.sendall(data)
            else:
                break
    finally:
        connection.close()
```

To see the actual address being used by a socket, call its `getsockname()` method. After starting the service, running `netstat` again shows it listening for incoming connections on any address.

```
$ netstat -an

Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address    Foreign Address  (state)
...
tcp4       0      0  *.10000          *.*              LISTEN
...
```

Addressing, Protocol Families and Socket Types

User Datagram Client and Server

Quick Links

Echo Server
Echo Client
Client and Server Together
Easy Client Connections
Choosing an Address for Listening

This page was last updated 2016-12-18.

Navigation

Addressing, Protocol Families and Socket Types
User Datagram Client and Server

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

- Blog
- The Python Standard Library By Example