

If you find this information useful, consider picking up a copy of my book, *The Python Standard Library By Example*.

---

## 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.

```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_I
```

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.

```
# Bind the socket to the port
server_address = ('localhost', 1
print >>sys.stderr, 'starting up
sock.bind(server_address)
```

Calling `listen()` puts the socket into server mode, and `accept()` waits for an incoming connection.

```
# Listen for incoming connection
sock.listen(1)

while True:
    # Wait for a connection
    print >>sys.stderr, 'waiting
    connection, client_address =
```

`accept()` returns an open connection between the server and client, along with

## Page Contents

- [TCP/IP Client and Server](#)
  - [Echo Server](#)
  - [Echo Client](#)
  - [Client and Server Together](#)
  - [Easy Client Connections](#)
  - [Choosing an Address for Listening](#)

## Navigation

### Table of Contents

**Previous:** [Addressing, Protocol Families and Socket Types](#)

**Next:** [User Datagram Client and Server](#)

---

## This Page

[Show Source](#)

---

## Examples

The output from all the example programs from PyMOTW has been generated with Python 2.7.8, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

---

---

---

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()`.

```
try:
    print >>sys.stderr, 'con

    # Receive the data in sn
    while True:
        data = connection.re
        print >>sys.stderr,
        if data:
            print >>sys.stde
            connection.senda
        else:
            print >>sys.stde
            break

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

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.

```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_I

# Connect the socket to the port
server_address = ('localhost', 1
print >>sys.stderr, 'connecting
sock.connect(server_address)
```

After the connection is established, data can be sent through the `socket` with `sendall()` and received with `recv()`, just as in the server.

```
try:
```

```

# Send data
message = 'This is the messag
print >>sys.stderr, 'sending
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 >>sys.stderr, 'received %d bytes' % len(data)

finally:
    print >>sys.stderr, 'closing socket'
    sock.close()

```

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 is:

```

$ python ./socket_echo_server.py

starting up on localhost port 10000
waiting for a connection
connection from ('127.0.0.1', 52000)
received "This is the mess"
sending data back to the client
received "age. It will be"
sending data back to the client
received " repeated."
sending data back to the client
received ""
no more data from ('127.0.0.1', 52000)
waiting for a connection

```

The client output is:

```

$ python socket_echo_client.py

connecting to localhost port 10000
sending "This is the message. I want to see if you can receive it."
received "This is the mess"
received "age. It will be"
received " repeated."
closing socket

$

```

## 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.

```
import socket
import sys

def get_constants(prefix):
    """Create a dictionary mapping socket constants to their names.
    return dict( (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(

print >>sys.stderr, 'Family :',
print >>sys.stderr, 'Type   :',
print >>sys.stderr, 'Protocol:',
print >>sys.stderr

try:

    # Send data
    message = 'This is the messa
    print >>sys.stderr, 'sending
    sock.sendall(message)

    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print >>sys.stderr, 'rec

finally:
    print >>sys.stderr, 'closing
    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.

```
$ python socket_echo_client_easy.py
Family : AF_INET
```

```
Type      : SOCK_STREAM
Protocol: IPPROTO_TCP

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

## 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.

```
import socket
import sys

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

# Bind the socket to the address
server_name = sys.argv[1]
server_address = (server_name, 1024)
print >>sys.stderr, 'starting up'
sock.bind(server_address)
sock.listen(1)

while True:
    print >>sys.stderr, 'waiting for'
    connection, client_address = sock.accept()
    try:
        print >>sys.stderr, 'client'
        while True:
            data = connection.recv(1024)
            print >>sys.stderr, 'received'
            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.

```
import socket
```

```

import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_I

# Connect the socket to the port
server_address = (sys.argv[1], 1
print >>sys.stderr, 'connecting
sock.connect(server_address)

try:

    message = 'This is the messa
    print >>sys.stderr, 'sending
    sock.sendall(message)

    amount_received = 0
    amount_expected = len(message)
    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print >>sys.stderr, 'rec

finally:
    sock.close()

```

After starting the server with the argument `farnsworth.hellfly.net`, the **netstat** command shows it listening on the address for the named host.

```

$ host farnsworth.hellfly.net

farnsworth.hellfly.net has address 192.168.1.1

$ netstat -an

Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address Foreign Address
...
tcp4      0      0 192.168.1.1:11111 0.0.0.0:*
...

```

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

```

$ hostname

homer

$ python socket_echo_client_expl

connecting to farnsworth.hellfly.net
sending "This is the message. I am a client."
received "This is the message. I am a client."
received "age. It will be repeated."

```

And the server output is:

```
$ python ./socket_echo_server_ex

starting up on farnsworth.hellfl
waiting for a connection
client connected: ('192.168.1.8'
received "This is the mess"
received "age. It will be"
received " repeated."
received ""
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 the empty string `''` instead of doing the conversion.

```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_I

# Bind the socket to the address
server_address = ('', 10000)
sock.bind(server_address)
print >>sys.stderr, 'starting up
sock.listen(1)

while True:
    print >>sys.stderr, 'waiting
    connection, client_address =
    try:
        print >>sys.stderr, 'cli
        while True:
            data = connection.re
            print >>sys.stderr,
            if data:
                connection.senda
            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 (incl  
Proto Recv-Q Send-Q Local Address  
...  
tcp4      0      0 *.10000  
...
```



## Comments Community

1 Login ▾

♥ Recommend Share

Sort by Best ▾

Start the discussion...

Be the first to comment.