Setting up a UDP Socket

We're now going to write some basic server code in TCP. Let's get right into it.

We'll cover the following

- Purpose of the Program
- Importing `socket`
- Creating a `socket` Object
- Binding The Socket
    - Port Number
    - Hostname

Remember that sockets are just software endpoints that processes write and read data from. They are bound to an IP address and a port. As we will see, the sending process attaches the IP address and port number of the receiving application. The IP address and port number of the sending process are also attached to the packets as headers, but that’s not done manually in the code of the application itself. Networking libraries are provided with nearly all programming languages and they take responsibility for lots of plumbing.

Purpose of the Program

Let’s set up a socket for a **UDP server program** that works like so:

1. The client will send a line of text to the server.
2. The server will receive the data and convert each character to **uppercase**.
3. The server will send the uppercase characters to the client.
4. The client will receive and display them on its screen.

We are building the code line-by-line to make it easier to understand. Moreover, we will highlight new additions at every step.

Importing `socket`

The first step when writing a network application in Python is to import the `socket` library. It's generally already part of the Python bundle, so no extra library will have to be manually installed.

```
import socket
```

Creating a `socket` Object

We now create a **socket object** called `s` with a call to `socket.socket()`.

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
print(s)
```

The syntax is as follows.

```
socket.socket(family, type, proto, fileno)
```

The syntax is shown here for completion, however, our main focus will be on explaining the **family** and **type** properties.

1. **Family**. The address family property is used to assign the type of addresses that a socket can communicate with. Only then can the addresses of that type be used with the socket. There are **three main options** available for this:

    - `AF_INET`. This family is used with **IPV4** addresses. When we first introduced IP addresses, we were talking about **IPV4** addresses. IP addresses are most commonly used.

    - `AF_INET6` Another address scheme, IPv6, was introduced since IPV4 is limited to about 4 billion addresses which are not sufficient, considering the exponential growth of the Internet. IPV6 provides 340 undecillion addresses (340×10^{36}). It's slowly being adopted. `AF_INET6` is used for IPV6 addresses. We'll introduce IPV6 more formally in a later lesson.

    - `AF_UNIX` This family is used for **Unix Domain Sockets (UDS)**, an interprocess communication endpoint for the same host. It's available on POSIX-compliant systems. Most operating systems today like Windows, Linux and Mac OS are POSIX compliant! So processes on a system can communicate with each other directly through this instead of having to communicate via the network.

2. **Type**. The type specifies the transport layer protocol:
    - `SOCK_DGRAM` specifies that the application is to use **User Datagram Protocol** (UDP). Recall that **UDP** is less reliable but requires no initial connection establishment. We are building these server and client programs pair in UDP.
    - `SOCK_STREAM` specifies that the application is to use **Transmission Control Protocol** (TCP). Recall that while TCP requires some initial setup, it's more reliable than UDP.

If you want to study the rest of the fields, have a look at the documentation. Default arguments are being used for the remaining arguments, which is fine in our example.

> **Note: INET socket vs UNIX socket** UNIX sockets are bound to a special file on your machine, whereas INET sockets are bound to an IP address and port. So UNIX sockets can only be accessed by the processes running on the machine and are therefore used for interprocess communication. INET sockets, on the other hand, can be accessed by remote machines.

Binding The Socket

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
port = 3000
hostname = '127.0.0.1'
s.bind((hostname, port))
```

Now, we bind the socket to an IP address and a port using the `bind()` function. It’s given a certain n-tuple as an argument, where `n` depends on the family of the socket. In the case of `SOCK_DGRAM`, it’s a tuple of the IP address and the port like the following.

```
socketObject.bind((IP address, port))
```

Port Number

We define the hostname and the port as variables on **lines 4 and 5**. The `port` is **3000** in this case, and as mentioned previously, the ports $0 - 1024$ should be avoided as they’re reserved for other system-defined processes. Binding to them may generate an error as it may already be in use by another application.

Hostname

The hostname is the IP address that your server will listen on. You can set it to one of three options:

1. If you’re following along on your local machine, set it to `127.0.0.1` which is the localhost address, for IPV4. This address is called the loopback or localhost address and you should use this because you’ll be writing the client code on the same machine as the server. We’re doing the same here.

2. You could also set it to the empty string `''` which represents the `INADDR_ANY`. This specifies that the program intends to receive packets sent to the specified port destined for any of the IP addresses configured on that machine.

3. Lastly, you could set it to a specific IP address assigned to your machine.

Now that we know how to set up a basic socket, which is necessary for any kind of network application using sockets, let’s write some code that is specific to servers.