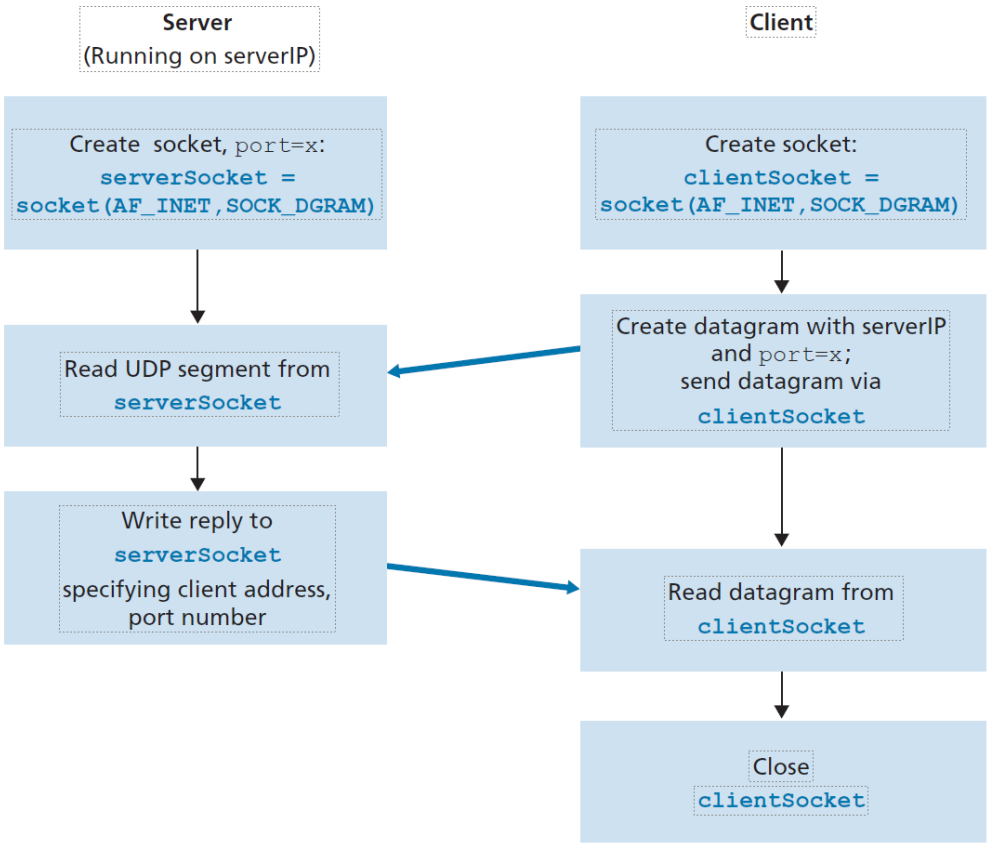
Socket programming Using UDP

Following figure highlights the main socket-related activity of the client and server that communicate over the UDP transport service.



1 Socket programming example: UDP server

We'll use the following simple client-server application to demonstrate socket programming for UDP:

1. The client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts the characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

1.1 Initialize

We import `socket` module which forms the basis of all network communications in Python. By including this line, we will be able to create sockets within our program.

```
In [ ]: from socket import *
```

We define the port that our server is listening by setting the integer variable `serverPort` to 12000.

```
In [ ]: serverPort = 12000
```

1.2 Create socket

We creates the server's socket, called `serverSocket`.

- The first parameter indicates the address family; in particular, `AF_INET` indicates that the underlying network is using IPv4 (will be discusses in Chapter 4).
- The second parameter indicates that the socket is of type `SOCK_DGRAM`, which means it is a UDP socket (rather than a TCP socket).

```
In [ ]: serverSocket = socket(AF_INET, SOCK_DGRAM)
```

We bind the port number `serverport` (12000) to the server's socket, explicitly. In this manner, when anyone sends a packet to port 12000 at the IP address of the server, that packet will be directed to this socket.

```
In [ ]: serverSocket.bind(('', serverPort))
        print("The server is ready to receive.")
```

1.3 Start a loop to wait for a packet to arrive

- When a packet arrives at the server's socket:
  - the packet's data is put into the variable `message`,
  - the packet's source address is put into the variable `clientAddress` which contains both the client's IP address and the client's port number.
- The server converts the `message` from bytes to a string by method `decode()`, uses the method `upper()` to capitalize it, and then puts it into the variable `modifiedMessage`.
- It then converts the capitalized message `modifiedMessage` from string type to byte type by method `encode()`, attaches the `clientAddress` (IP address and port number of the client) to the message, and sends the resulting packet into the server's socket. The Internet will then deliver the packet to this client address.
- After the server sends the packet, it remains in the while loop, waiting for another UDP packet to arrive or Ctrl-C to be pressed on the keyboard.

```
In [ ]: try:
            while True:
                message, clientAddress = serverSocket.recvfrom(2048)
                modifiedMessage = message.decode().upper()
```

```
            serverSocket.sendto(modifiedMessage.encode(), clientAddress)
    except KeyboardInterrupt:
        print("Press Ctrl-C to terminate while statement")
        pass
```

2 Socket programing example: UDP client

We'll use the following simple client-server application to demonstrate socket programming for UDP:

1. The client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts the characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

2.1 Initialize

We import `socket` module which forms the basis of all network communications in Python. By including this line, we will be able to create sockets within our program.

```
In [ ]: from socket import *
```

We define the address and port that our server is listening by setting the integer variables `serverName` to "127.0.0.1" (localhost) and `serverPort` to 12000, respectively.

```
In [ ]: serverName = "127.0.0.1"
        serverPort = 12000
```

2.2 Create socket

We creates the client's socket, called `clientSocket`.

- The first parameter indicates the address family; in particular, `AF_INET` indicates that the underlying network is using IPv4 (will be discusses in Chapter 4).
- The second parameter indicates that the socket is of type `SOCK_DGRAM`, which means it is a UDP socket (rather than a TCP socket).
- Note that we are again not specifying the port number of the client socket when we create it; we are instead letting the operating system do this for us.

```
In [ ]: clientSocket = socket(AF_INET, SOCK_DGRAM)
```

2.3 Read user message

`input` method will promt the words "Input lowercase sentence:", and then waits for an user input, which is put into the variable `message`.

```
In [ ]: message = input("Input lowercase sentence:")
```

2.4 Send the message to server

- We first convert the `message` from string type to byte type, as we need to send bytes into a socket; this is done with the `encode()` method.
- The method `sendto()` attaches the destination address `(serverName, serverPort)` to the message and sends the resulting packet into the process's socket, `clientSocket`.

```
In [ ]: clientSocket.sendto(message.encode(), (serverName, serverPort))
```

2.5 Receive modified message from server

- When a packet arrives at the client's socket:
  - the packet's data is put into the variable `modifiedMessage`,
  - the packet's source address is put into the variable `serverAddress` which contains both the server's IP address and the server's port number.
- The method `recvfrom` takes the buffer size 2048 as input (this buffer size works for most purposes.).

```
In [ ]: modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
```

2.6 Print the modified message and finish

- The client converts the `modifiedMessage` from bytes to a string by method `decode()`, and then prints out it on the display.
- It then closes the socket. The process terminates.

```
In [ ]: print(modifiedMessage.decode())
        clientSocket.close()
```