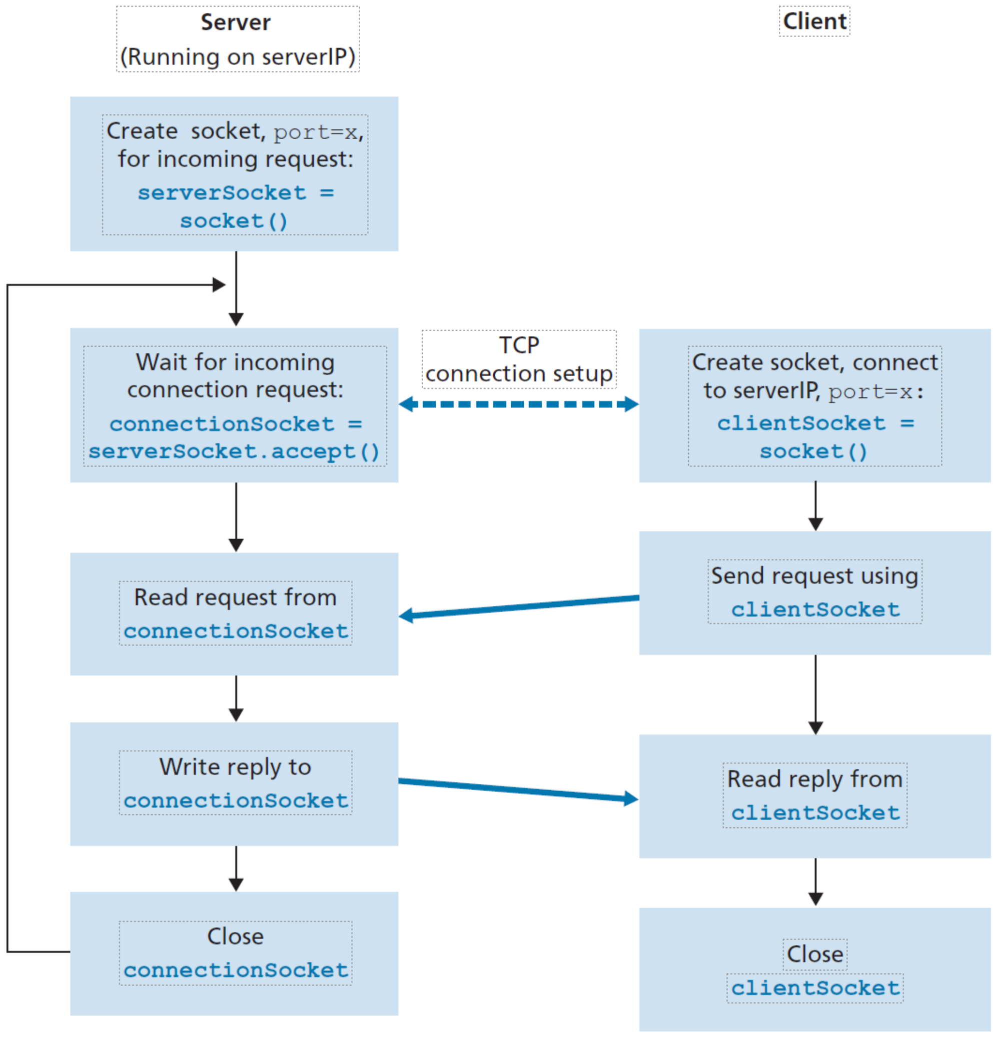
Socket Programming Using TCP

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


Server
(Running on serverIP)
Client
Create socket, port=x, for incoming request:
serverSocket = socket()
Wait for incoming connection request:
connectionSocket = serverSocket.accept()
TCP connection setup
Create socket, connect to serverIP, port=x:
clientSocket = socket()
Send request using clientSocket
Read request from connectionSocket
Write reply to connectionSocket
Read reply from clientSocket
Close connectionSocket
Close clientSocket

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

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 Socket programing example: TCP server

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_STREAM, which means it is a TCP socket (rather than a UDP socket).

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

1.3 Bind with a port and listen

We bind the port number serverport (12000) to the server's socket, explicitly.

```
In [ ]: serverSocket.bind(('', serverPort))
```

The server listen for TCP connection requests from the client. The parameter specifies the maximum number of queued connections (at least 1)

```
In [ ]: serverSocket.listen(1)
        print("The server is ready to receive.")
```

1.4 Start a loop to wait for a client connection and its message

- When a connection request arrives at the server's socket:
  - The program invokes the accept() method for serverSocket, which creates a new socket in the server, called connectionSocket, dedicated to this particular client.
  - The request's source address is put into the variable clientAddress which contains both the client's IP address and the client's port number.
  - The client and server then complete the hand-shaking, creating a TCP connection between the client's clientSocket and the server's connectionSocket.
  - With the TCP connection established, the client and server can now send bytes to each other over the connection. With TCP, all bytes sent from one side are only guaranteed to arrive at the other side but also guaranteed to arrive in order.

- When a character arrives at the `connectionSocket`:
  - The character is put into the variable `message`.
  - The server continue to accumulate characters from client in `message` until the line ends with a carriage return character.
  - The method `recv` takes the buffer size 2048 as input (this buffer size works for most purposes.).
- 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()` and sends the bytes into the `connectionSocket`.
- After sending the modified sentence to the client, the program closes the connection socket.
- After the server closes the socket, it remains in the while loop, waiting for another client connection or Ctrl-C to be pressed on the keyboard.

```
In [ ]: try:
            while True:
                connectionSocket, clientAddress = serverSocket.accept()
                message = connectionSocket.recv(2048)
                modifiedMessage = message.decode().upper()
                connectionSocket.send(modifiedMessage.encode())
                connectionSocket.close()
        except KeyboardInterrupt:
            print("Press Ctrl-C to terminate while statement")
            pass
```

2 Socket programing example: TCPclient

2.1 Initiallize

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 [1]: 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 [2]: 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_STREAM`, which means it is a TCP socket (rather than a UDP 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 [3]: clientSocket = socket(AF_INET, SOCK_STREAM)
```

2.3 Connect to server

```
In [4]: clientSocket.connect((serverName,serverPort))
```

2.4 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 [5]: message = input("Input lowercase sentence:")
```

```
Input lowercase sentence:hello
```

2.5 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.
- Note that the program does not explicitly create a packet and attach the destination address to the packet, as was the case with UDP sockets. Instead the client program simply sends the bytes in the string `message` over the TCP connection through the client's socket, `clientSocket`, by the method `send()`.

```
In [6]: clientSocket.send(message.encode())

Out[6]: 5
```

2.6 Receive modified message from server

- When a character arrives at the client's socket, it is put into the variable `modifiedMessage`,
- The client continue to accumulate characters from server in `modifiedMessage` until the line ends with a carriage return character
- The method `recv` takes the buffer size 2048 as input (this buffer size works for most purposes.).

```
In [7]: modifiedMessage = clientSocket.recv(2048)
```

2.7 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 [8]: print(modifiedMessage.decode())
        clientSocket.close()
```