T.C.

MARMARA UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

2019 – 2020 FALL

CSE4074 – COMPUTER NETWORKS

SOCKET PROGRAMMING – HTTP SERVER AND PROXY SERVER PROJECT

150114022

OĞUZHAN BÖLÜKBAŞ

- 07.12.2018 –

Socket Programming

Typical network applications consist of a pair of programs residing in two different end systems, a client program and a server program. When these two programs are executed, a client process and a server process are created, and these processes communicate with each other by reading from, and writing to, sockets. When creating a network application, the developer's main task is therefore to write the code for both the client and server programs. There are two types of network applications. One type is an implementation whose operation is specified in a protocol standard, such as an RFC or some other standards document; such an application is sometimes referred to as "open," since the rules specifying its operation are known to all.

For such an implementation, the client and server programs must conform to the rules dictated by the RFC. For example, the client program could be an implementation of the client side of the HTTP protocol and precisely defined in RFC 2616; similarly, the server program could be an implementation of the HTTP server protocol, also precisely defined in RFC 2616. If one developer writes code for the client program and another developer writes code for the server program, and both developers carefully follow the rules of the RFC, then the two programs will be able to interoperate. Indeed, many of today's network applications involve communication between client and server programs that have been created by independent developers—for example, a Google Chrome browser communicating with an Apache Web server, or a BitTorrent client communicating with BitTorrent tracker.

The other type of network application is a proprietary network application. In this case the client and server programs employ an application-layer protocol that has not been openly published in an RFC or elsewhere. A single developer (or development team) creates both the client and server programs, and the developer has complete control over what goes in the code. But because the code does not implement an open protocol, other independent developers will not be able to develop code that interoperates with the application.

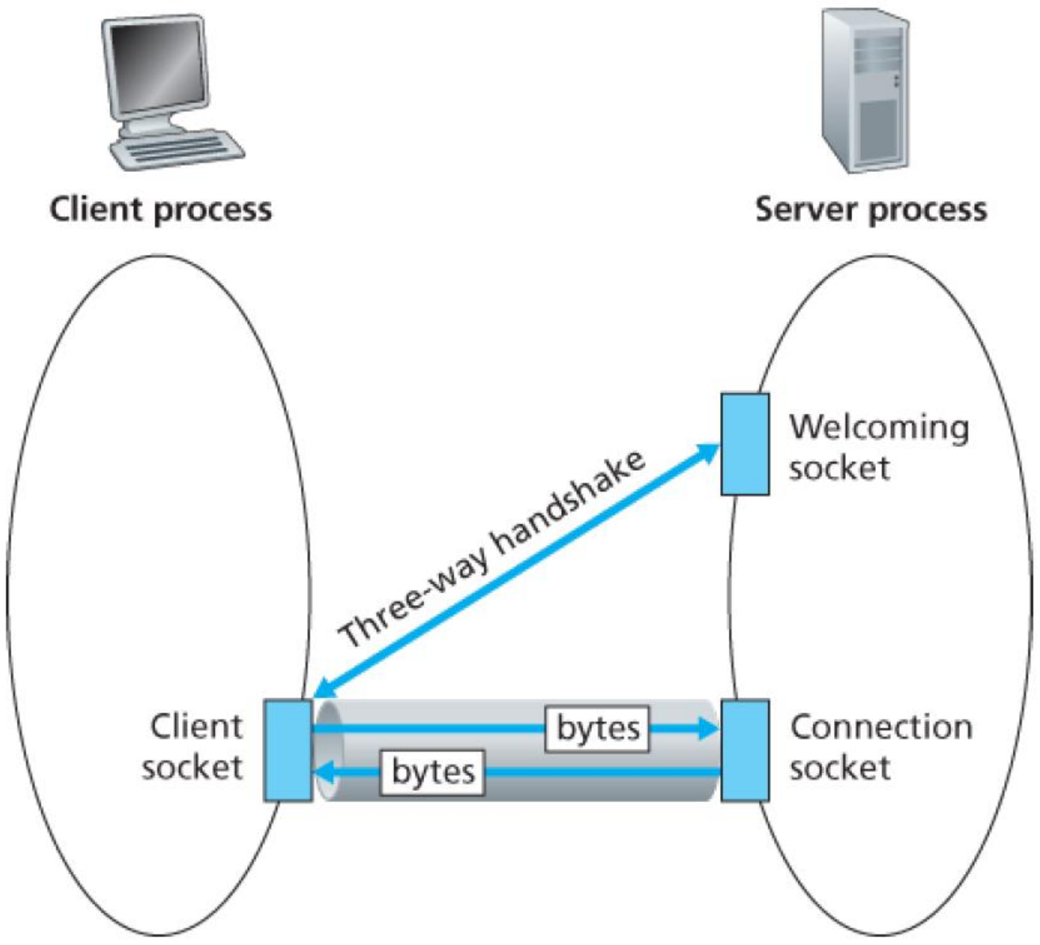
Socket Programming with TCP

TCP is a connection-oriented protocol. This means that before the client and server can start to send data to each other, they first need to handshake and establish a TCP connection. One end of the TCP connection is attached to the client socket and the other end is attached to a server socket.

When creating the TCP connection, we associate with it the client socket address (IP address and port number) and the server socket address (IP address and port number). With the TCP connection established, when one side wants to send data to the other side, it just drops the data into the TCP connection via its socket. This is different from UDP, for which the server must attach a destination address to the packet before dropping it into the socket.

The interaction of client and server programs in TCP, the client has the job of initiating contact with the server. For the server to be able to react to the client's initial contact, the server must be ready. This implies two things. First, as in the case of UDP, the TCP server must be running as a process before the client attempts to initiate contact. Second, the server program must have a special door—more precisely, a special socket—that welcomes some initial contact from a client process running on an arbitrary host. Using our house/door analogy for a process/socket, we will sometimes refer to the client's initial contact as "knocking on the welcoming door."

With the server process running, the client process can initiate a TCP connection to the server. This is done in the client program by creating a TCP socket. When the client creates its TCP socket, it specifies the address of the welcoming socket in the server, namely, the IP address of the server host and the port number of the socket. After creating its socket, the client initiates a three-way handshake and establishes a TCP connection with the server. The three-way handshake, which takes place within the transport layer, is completely invisible to the client and server programs.

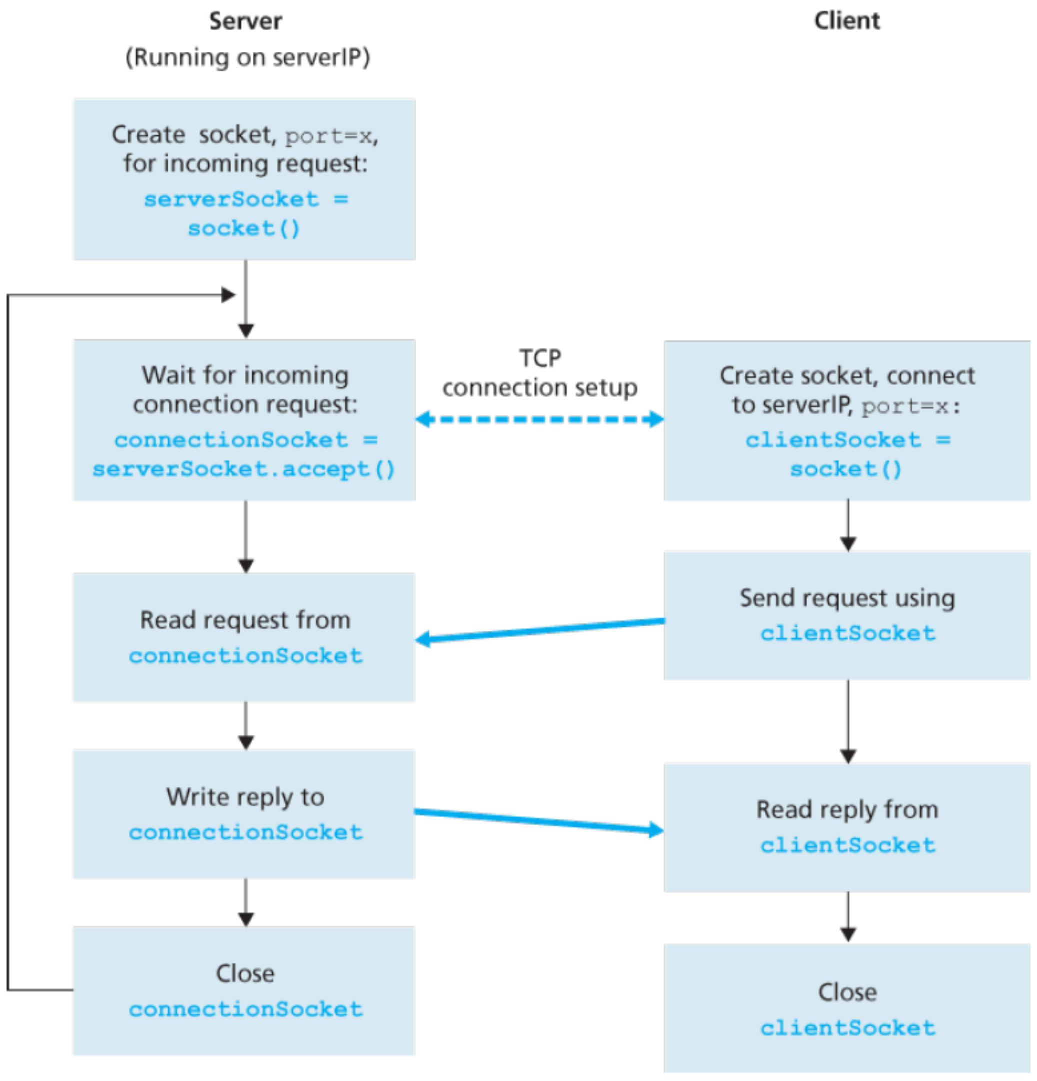


[1]

Figure 1: The TCP Server process has two sockets

Here is the code for the **client** side of the application in Python:

```
from socket import *
serverName = ’servername’
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input(’Input lowercase sentence:’)
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print(’From Server: ’, modifiedSentence.decode())
clientSocket.close()
```



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

[1]

Figure 2: The client-server application using TCP

Here is the code for the **server** side of the application in Python:

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind((’’, serverPort))
serverSocket.listen(1)
print(’The server is ready to receive’)
while True:
   connectionSocket, addr = serverSocket.accept()
   sentence = connectionSocket.recv(1024).decode()
   capitalizedSentence = sentence.upper()
   connectionSocket.send(capitalizedSentence.encode())
   connectionSocket.close()
```

Implementation

Architecture of the main program is below

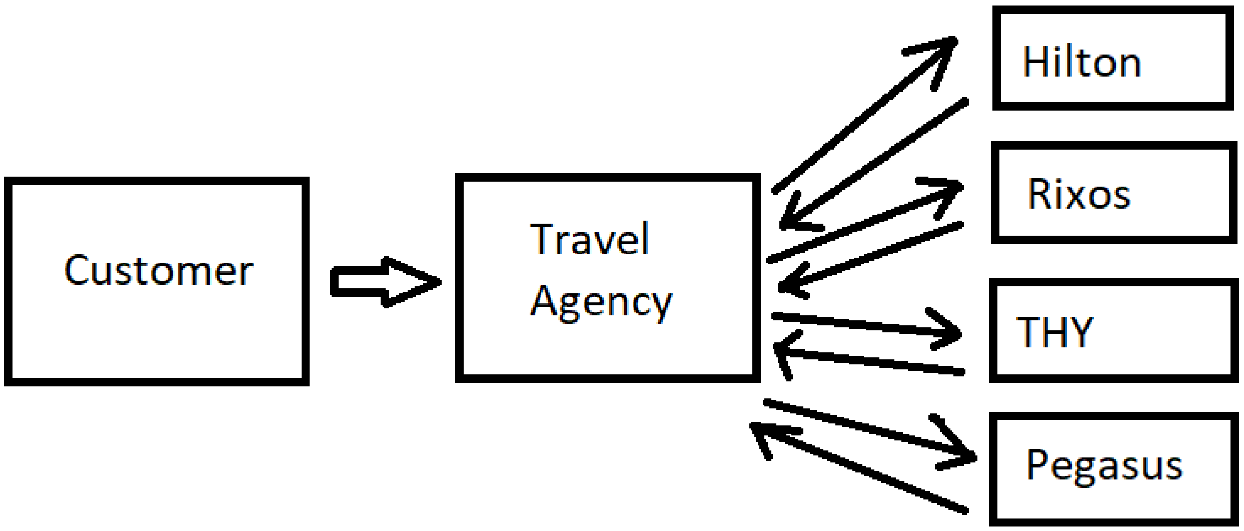


Figure 1 - Architecture of the project

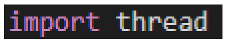
Customer part is running on **“customer.py”** file. It takes inputs from user via GUI. After taking input, it sends information to travel agency.

Travel agency part is running on “**agency.py**” file. After obtaining travel information, it sends it to desired hotel and airline company. These companies make reservations if they have enough capacity. If they do not have, they check capacities for different days. If they found any suitable, they suggest these days to customer via GUI.

Multithread bonus

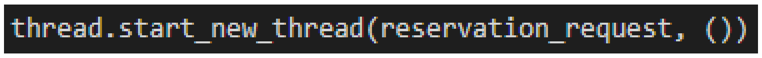
Travel agency runs reservations wishes in multithreading programming way. It runs required methods via threads, so it can response other customer while previous customer process is on running.

Multithread library runs on Python 2. So I have written my project in Python 2. Multithread library is:



```
import thread
```

To run program with threads, run "**start_new_thread**" function which is placed into thread library with giving function and function's argument as **start_new_thread**'s arguments



```
thread.start_new_thread(reservation_request, ())
```

Dynamic programming bonus

When all capacities are fulfilled, travel agency makes a file to request new hotel and airline companies. Hotel and airlines companies writes their information into a file and save them after seeing travel agency's desire. Before this, they do not serve.