ELEC-C7420 Basic Principles in Networking

Introduction to Socket Programming

Tutorial Session (07-02-2022)

Prerequisites

- Networking concepts (client-server model, addressing the network)
- Basic knowledge of python

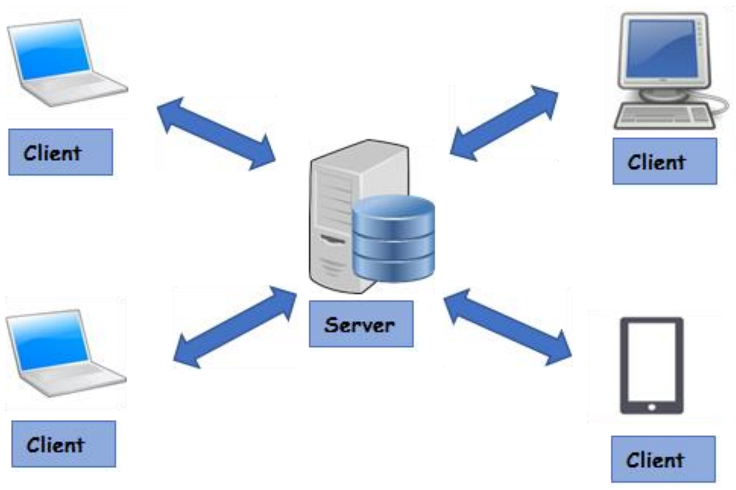
What is networking?

Networking, also known as computer networking, is the practice of transporting and exchanging data between nodes over a shared medium in an information system. Networking comprises not only the design, construction and use of a network, but also the management, maintenance and operation of the network infrastructure, software and policies. It is a concept of two programs communicating across a network. Whether it be from *client-client*, *client-server* or even *client to itself*.

Client:	An end device interfacing with a human
Server:	A device providing a service for the clients.

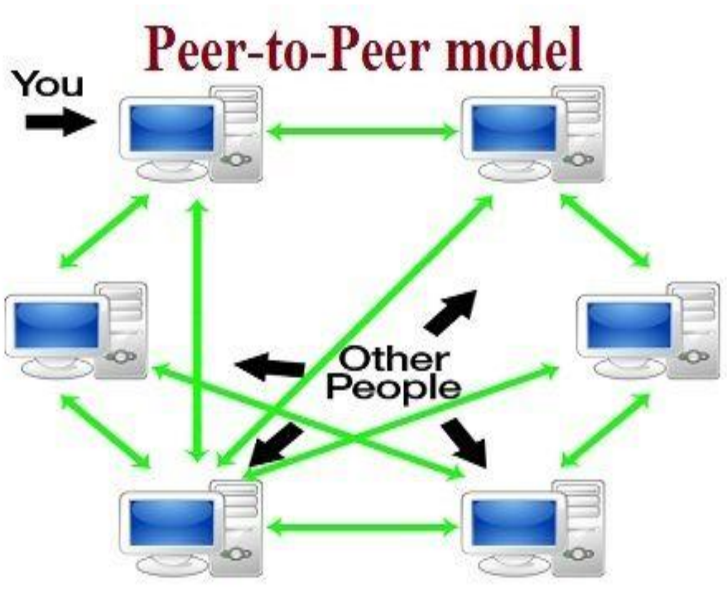
Common Networking Models

i. Client/Server Model



- Most Common
- Clients connect to the server to get the information they require
- Web browser (Client) connects to the Google website (Server).

ii. Peer-to Peer Model



- Useful for service that don't have to be constantly available (Skype, Game Servers etc.)
- Clients connect to the clients without the use of a central server.
- Is actually a Client/Server model at its core, just clients are acting as a server and client

Terminology

- Address:
  - ☐ An IPv4 address, e.g., 127.0.0.1 (look back address, points back to your own computer)
  - ☐ An IPv6 address, e.g., 2001::0:735c:19b7:fde:b8b3:2d9b.
- Port: A port number, e.g., 80 (Port number 1-1024 are reserved for core protocols. Try to use something above 1024 and below 65535).

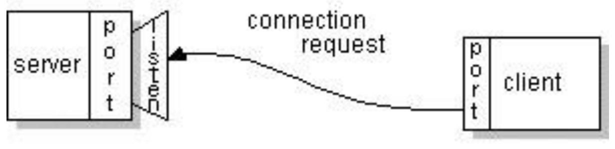
What is TCP?

- Transmission Control Protocol (TCP).
- Reliable Connection Based Protocol. Forms the connections with other device and keeps the connection going until it is closed.
- Ordered & Error checked (simple checksum). If it arrives in an ordered manner. TCP is slower than other protocols due to all these checks and making sure all the data is there. It is used in programs where reliable transmission is required.
- Used by Web browsers, Email, SSH, FTP etc.

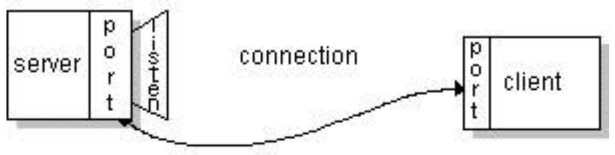
What is *socket*?

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's hostname and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created, and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Definition:

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to an address and to a port number so that the TCP layer can identify the application that data is destined to be sent to.

To summarize, *socket*

- Sockets are the programming abstraction for connections
- They allow us to communicate in a bidirectional manner
- Once they are connected or ready to transmit, we can use them to send and receive data.
- They implement the common transport protocols like TCP and UDP.

Further considerations:

Before doing realistic applications with many devices, don't forget to allow connections to ports through firewalls and NATs.

Socket Methods

To create a socket, you must use the `socket.socket()` function available in `socket` module, which has the general syntax

```
s = socket.socket(family=AF_INET, type=SOCK_STREAM)
```

Here is the description of the parameters

- `socket_family` - This is by default AF_INET, meaning the port and address are given in a tuple `(host, port)`. AF_INET6 for IPv6.
- `socket_type` - This is usually SOCK_STREAM (TCP) or SOCK_DGRAM (UDP).

Once you have a `socket` object, then you can use functions to create your client or server program. Following is the list of the required functions

Server Socket Methods

- `s.bind((hostname,port))` – Tuple of a host address & port.
- `s.listen()` – Start listening to incoming TCP connections.
- `s.accept()` – Accepts a connection when found. Returns `(conn, address)`, where conn is new socket used to communicate with the client.

Client Socket Methods

- `s.connect((hostname,port))` – From a client side, to request a connection to a listening server.

General Socket Methods

- `s.recv(bufsize)` – Tries to grab data from a TCP connection. (Waits)
- `s.recvfrom(bufsize)` – Tries to grab data from a UDP connection.
- `s.send(bytes)` – Attempts to send the data given to it (TCP).
- `s.sendto(bytes, addr)` – Attempts to send the data given to it (UDP).
- `s.close()` – Closes a socket/connection and frees the port.
- `socket.gethostname()` – Returns the hostname of local machine

Demonstration of Socket Programming

Application Requisites:

- Python client (IDLE, Kivy, PyQt, PyCharm, etc.) for Windows.
- Gedit/vi/nano editor for Linux.

An example of a script for connecting to Google:

```
# An example script to connect to Google using socket
# programming in Python

import socket                                           # for socket

s = socket.socket()
print("Socket successfully created")

port = 80                                               # default port for socket
host_ip = socket.gethostbyname('www.google.com')        # get the server address
s.connect((host_ip, port))                             # connecting to the server

print("the socket has successfully connected to google on
address == %s" %(host_ip))
```

Output:

```
Socket successfully created
the socket has successfully connected to google on
address == 172.217.21.132
```

In the above demonstration we created a socket. Then we resolved Google's IP and finally, we connected to Google. Now we can go for sending some data through a socket.

A Simple Server Demonstration

To write Internet servers, we use the **socket** function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call `bind(hostname, port)` function to specify a *port* for your service on the given host.

Next, call the *accept* method of the returned object. This method waits until a client connects to the port you specified, and then returns a *connection* object that represents the connection to that client.

```
#!/usr/bin/python                    # This is server.py file

import socket                        # Import socket module

s = socket.socket()                  # Create a socket object
host = socket.gethostname()          # Get local machine name
port = 12345                         # Reserve a port for your service.
s.bind((host, port))                 # Bind to the port

s.listen(5)                          # Now wait for client connection.
while True:
   c, addr = s.accept()              # Establish connection with client.
   print('Got connection from', addr)
   c.send('Thank you for connecting'.encode())
   c.close()                         # Close the connection
```

A Simple Client Demonstration

Let us write a very simple client program which opens a connection to a given port 12345 and given host. It is very simple to create a socket client using Python's *socket* module function.

The `socket.connect((hostname, port))` opens a TCP connection to *hostname* on the *port*. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits

```
#!/usr/bin/python                    # This is client.py file

import socket                        # Import socket module
s = socket.socket()                  # Create a socket object
host = socket.gethostname()          # Get server host name (localhost)
port = 12345                         # This is the port of your server

s.connect((host, port))              # Connect to server
print(s.recv(1024).decode())         # Print message from server
s.close()                            # Close the socket when done
```

Now run this server.py and then run client.py to see the result.

```
# Following would start the server:
$ python server.py

# Once server is started run client as follows:
$ python client.py
```

This would produce following results

```
# Server:
Got connection from ('127.0.0.1', 48437)

# Client:
Thank you for connecting
```

Client is connected to the server at the given hostname & port and server sends the connection confirmation over the socket to the client.

IPv6 sockets

If you wish to use IPv6 in your program instead, you need to specify that the socket is using IPv6:

```
s = socket.socket(family=socket.AF_INET6)
```

An IPv6 socket can be run on the same port as an IPv4 port without conflicts. This is not the case for two IPv4 or IPv6 sockets.

This was a quick start with socket programming. Sockets are flexible and sufficient. Efficient socket-based programming can be easily implemented for general communications. Sockets cause low network traffic. Over the upcoming weeks, we will go deeper on socket programming and try to understand the usefulness of *socket* by implementing some exercise works.