# Socket Programming in Python

ECE 358 F22 Lab 2 Tutorial

Yiqing Irene Huang

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

# Learning Objectives

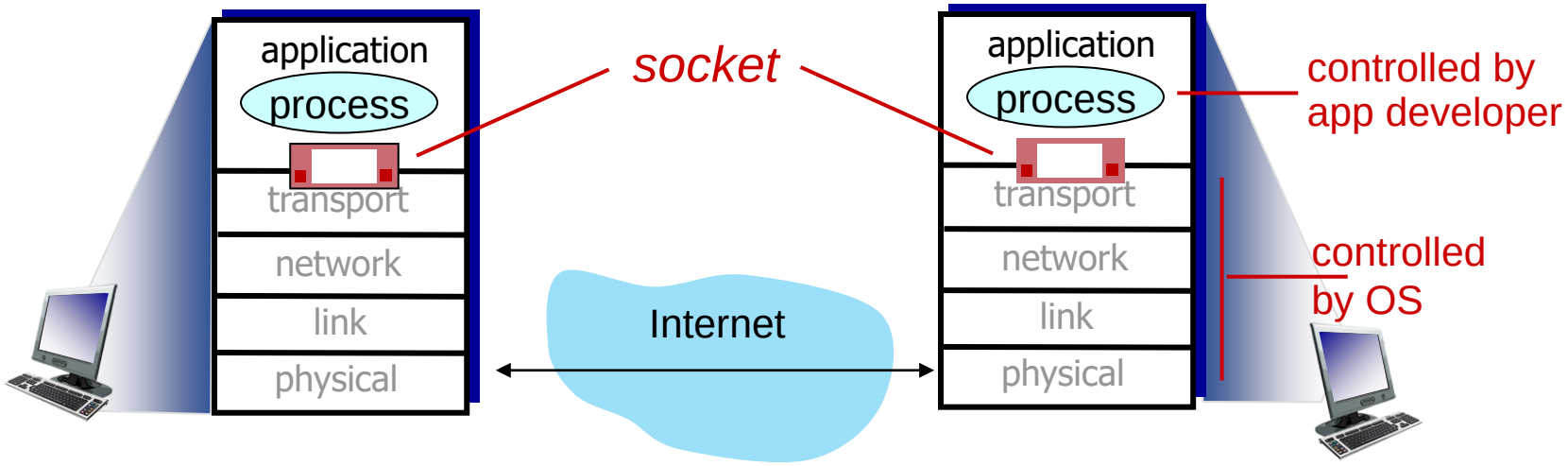
- Goal: learn how to build client/server applications that communicate using sockets in python
- Task 1: a web server using TCP to communicate with a client
  - HTTP protocol message format
- Task 2: an authoritative DNS server client system using UDP
  - DNS protocol message format
- Task 3: pencil and paper questions

# What is a Socket

- It is a UNIX interface for remote process communication developed by Berkeley in 1980s
- It is an abstract data structure as an endpoint of a communication link that supports full duplex information exchange.
- It supports TCP/IP and UDP protocols. It is widely used as a primitive for communication over Internet
- The interface is a set of system calls (C routines)
- Newer languages: JAVA, python, et. al..

# Socket programming

socket: door between application process and end-end-transport protocol



# Socket Type Classification

- Stream socket
  - Reliable (packets received in order)
  - Connection-oriented communication
  - Less efficient
  - TCP as the default protocol
- Datagram socket
  - Unreliable (packets received out of order)
  - Connectionless communication
  - More efficient
  - UDP as default protocol

# Socket programming

## Two socket types for two transport services:

- TCP*:* reliable, byte stream-oriented
- UDP*:* unreliable datagram

## Application Example:

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

# Socket programming with TCP

## Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

## Client contacts server by:

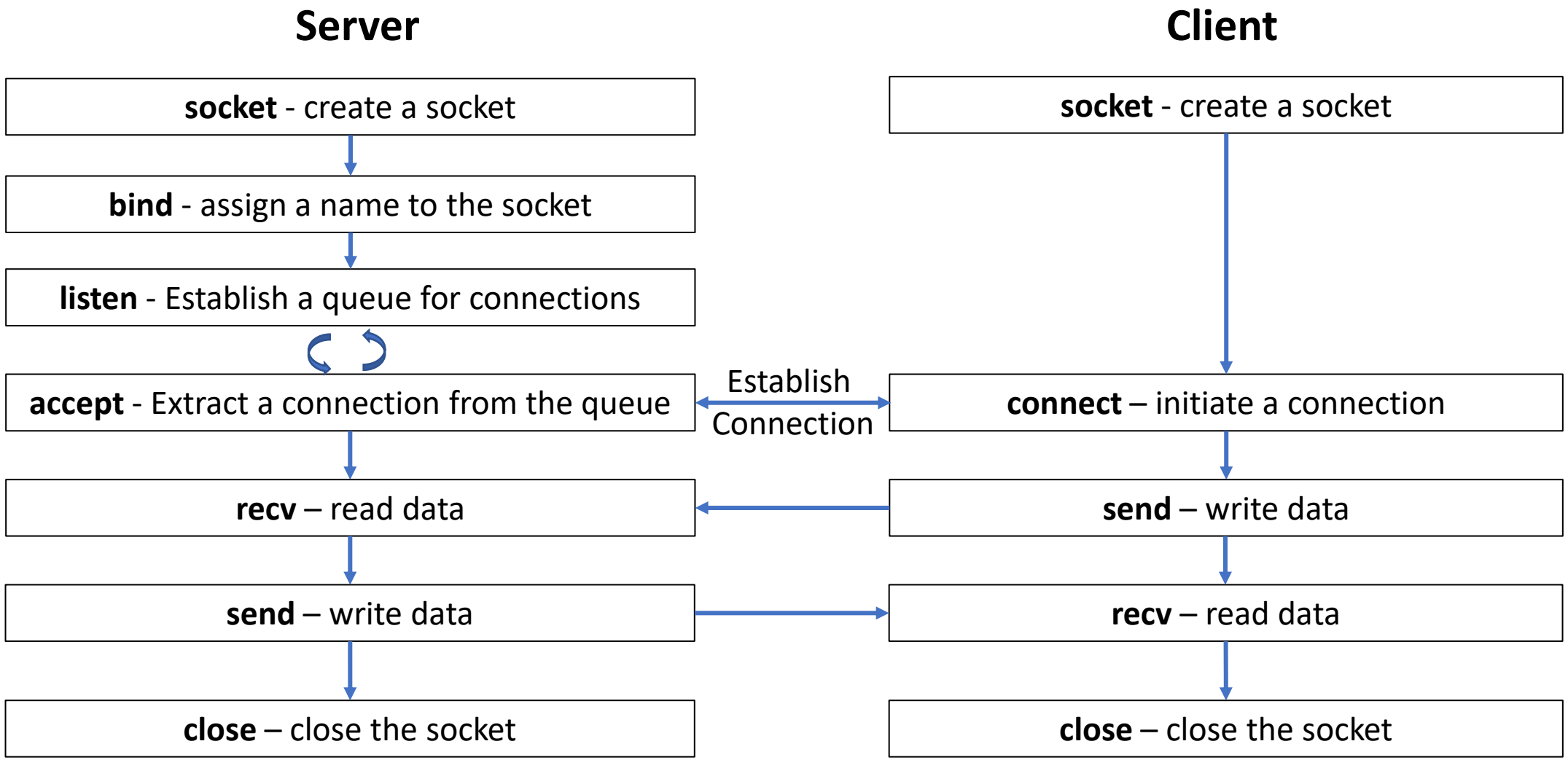
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - *source* port numbers used to distinguish clients

**Application viewpoint**

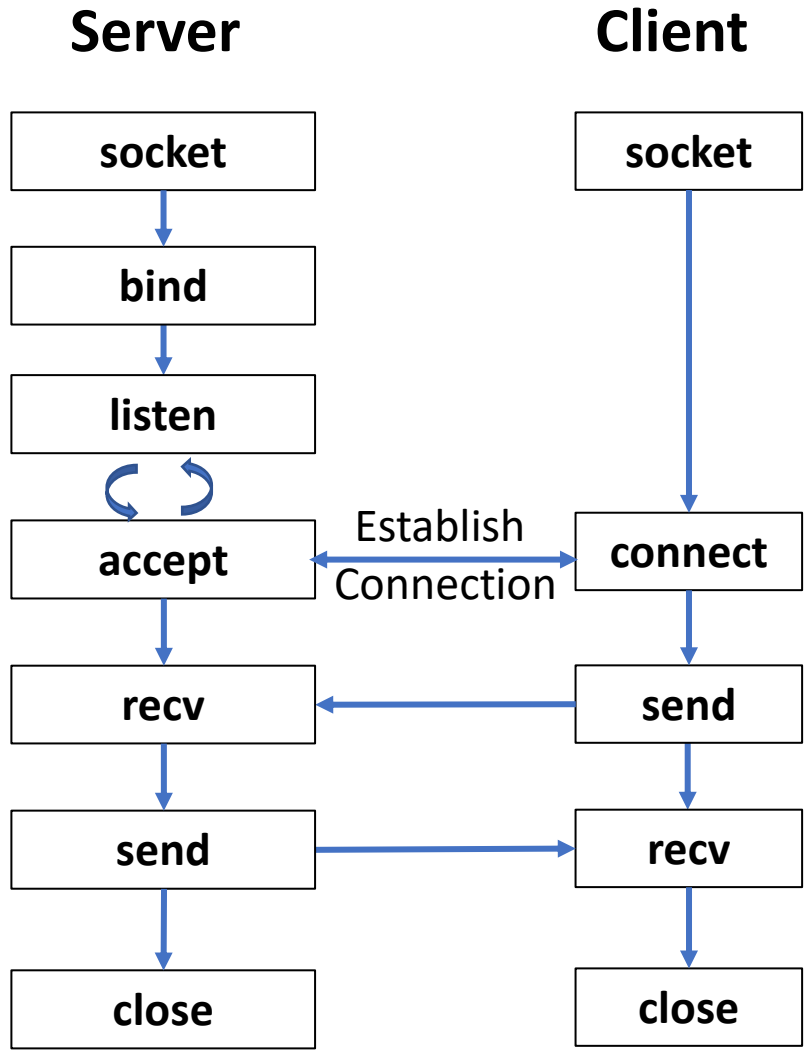
TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server processes

# TCP Socket Flow



# TCP Server in Python

```
from socket import *
serverIP="127.0.0.1"
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((serverIP, serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(2048).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
    connectionSocket.close()
```



# TCP Server – Commented Python Code

## Python TCP Server

```
 from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('127.0.0.1',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(2048).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
    connectionSocket.close()
```

create TCP welcoming socket →

server begins listening for incoming TCP requests →

loop forever →

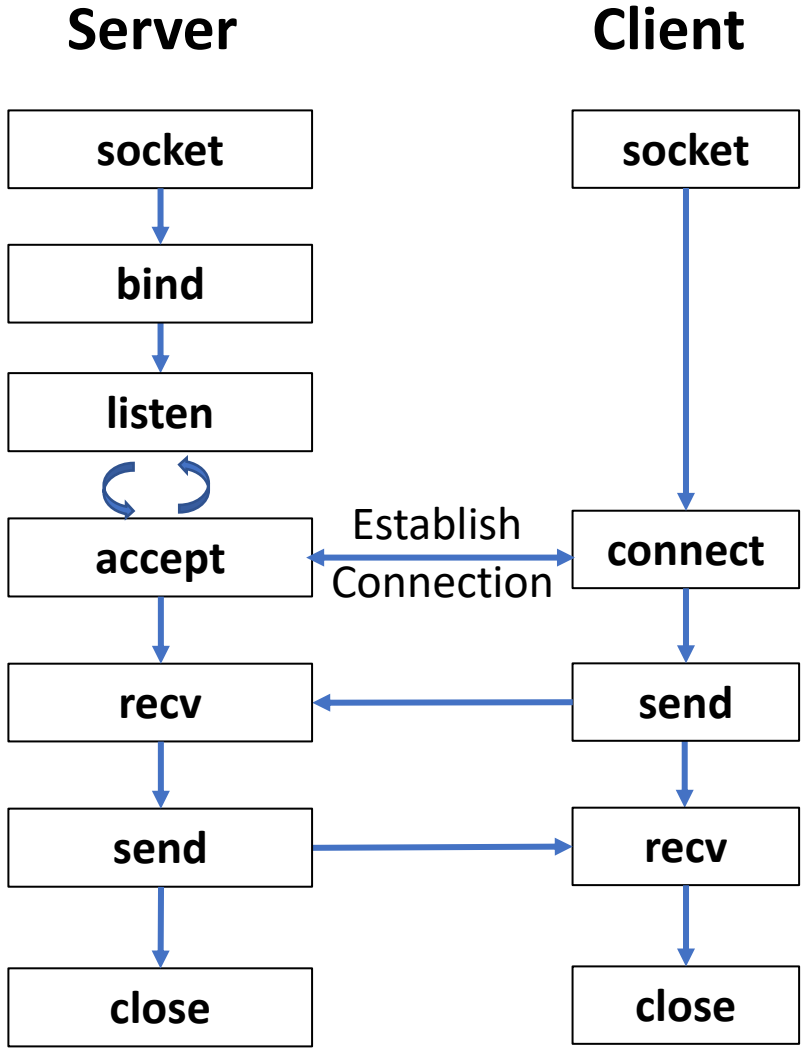
server waits on accept() for incoming requests, new socket created on return →

read bytes from socket (but no address as in UDP) →

close connection to this client (but *not* welcoming socket) →

# TCP Client in Python

```
from socket import *
serverIP = "127.0.0.1"
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverIP,serverPort))
sentence = input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(2048)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```



# TCP client – Commented Python Code

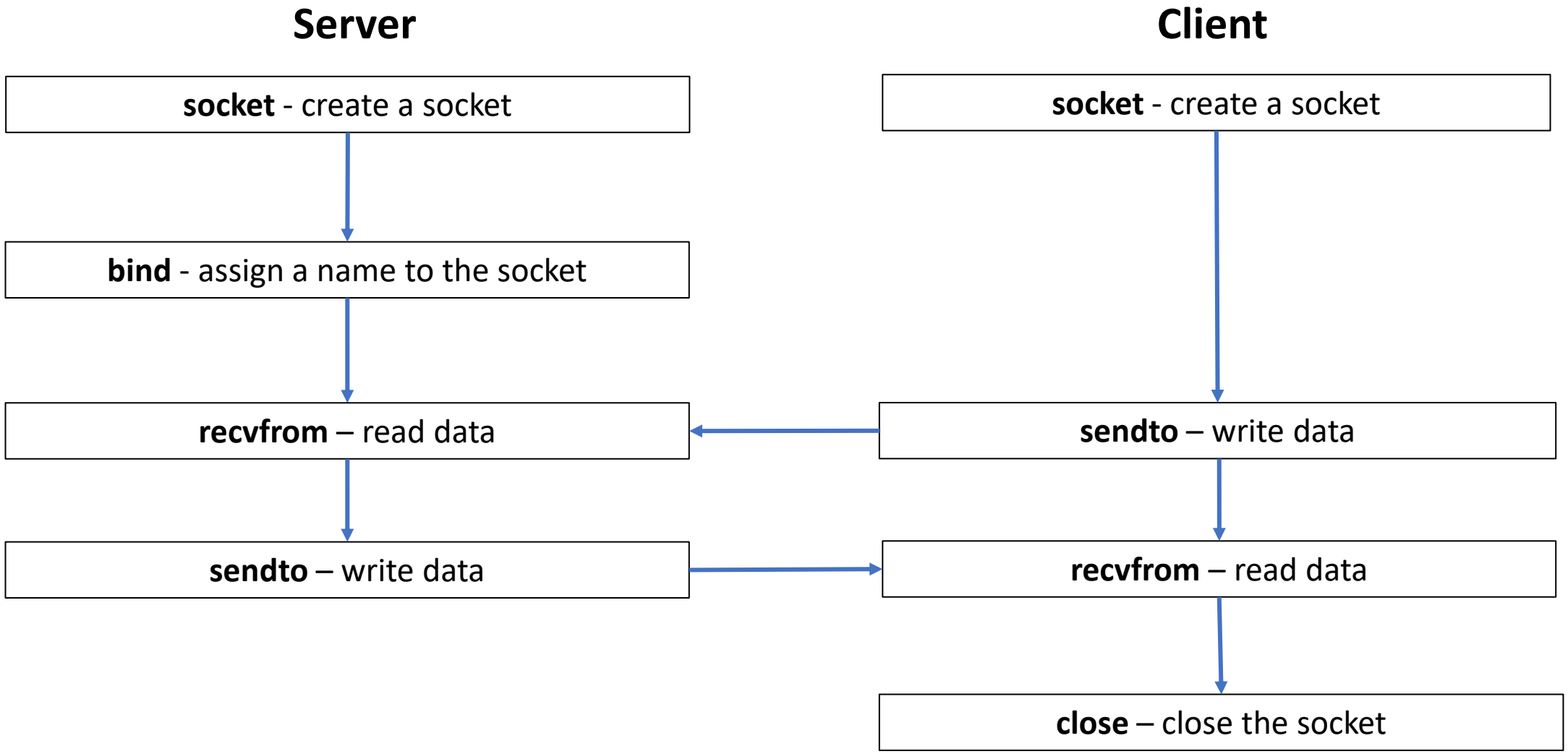
Python TCP Client

```
from socket import *
serverName = "127.0.0.1"
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(2048)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

create TCP socket for server, remote port 12000 →

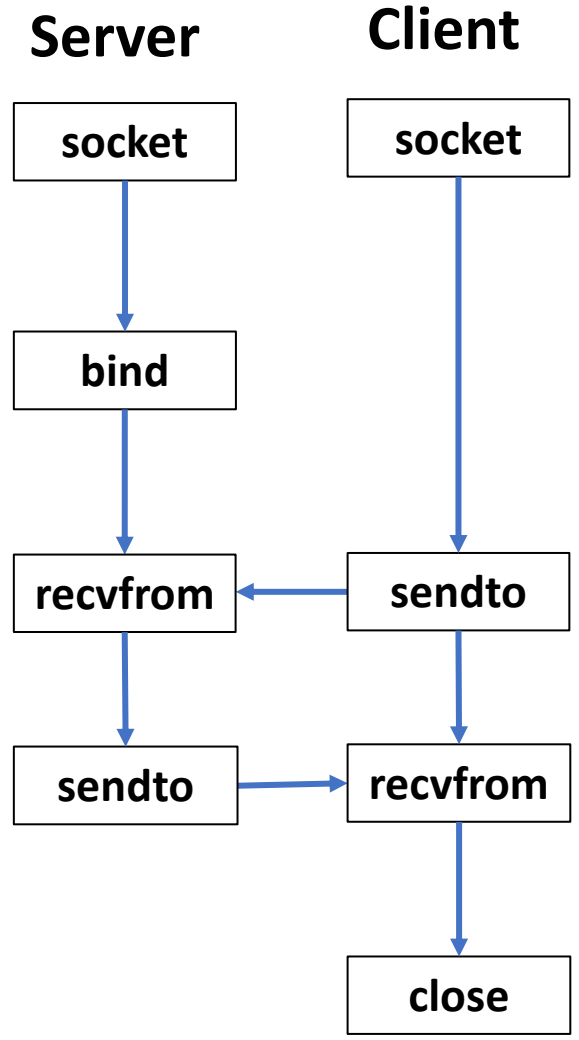
No need to attach server name, port →

# UDP Socket Flow



# UDP Server in Python

```
from socket import *
serverIP = "127.0.0.1"
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind((serverIP, serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),clientAddress)
```



# UDP Server – Commented Python Code

## Python UDP Server

```
from socket import *
serverIP = "127.0.0.1"
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),clientAddress)
```

create UDP socket →

bind socket to local port number 12000 →

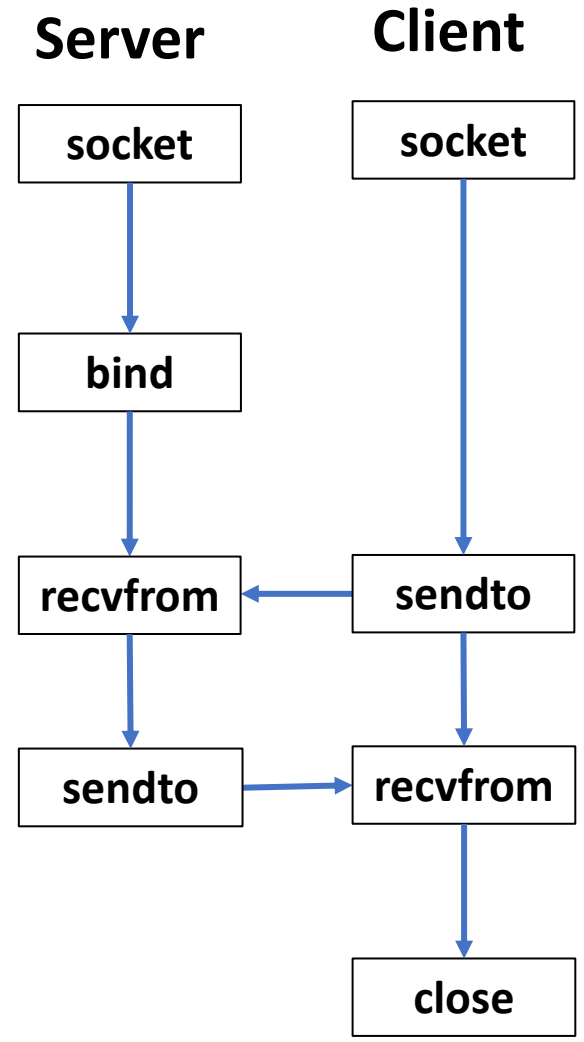
loop forever →

Read from UDP socket into message, getting client's address (client IP and port) →

send upper case string back to this client →

# UDP Client in Python

```
from socket import *
serverIP = '127.0.0.1'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input('Input lowercase sentence:')
clientSocket.sendto(message.encode(), (serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```



# UDP Client – Commented Python Code

## Python UDP Client

include Python's socket library →
```
from socket import *
```
```
serverIP = "127.0.0.1"
```
```
serverPort = 12000
```
create UDP socket for server →
```
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
```
get user keyboard input →
```
message = input('Input lowercase sentence:')
```
attach server name, port to message; send into socket →
```
clientSocket.sendto(message.encode(),
                    (serverIP, serverPort))
```
read reply characters from socket into string →
```
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
```
print out received string and close socket →
```
print modifiedMessage.decode()
```
```
clientSocket.close()
```

# Web Server

- TCP Socket Programming
- HTTP Protocol (https://datatracker.ietf.org/doc/html/rfc2616 )
- Requirements
  - Handles GET and HEAD requests
  - Response messages
    - Minimum 6 header lines
      Connection, Date, Server, Last-Modified, Content-Length, and Content-Type
  - The html files structures start from the directory where the server is in
- Use loopback address 127.0.0.1 and any non-privileged port number. Port number range [10000, 11000] is recommended.
- Client query URL example: http://127.0.0.1:6789/HelloWorld.html

# An Authoritative DNS Server and Client

- UDP Socket Programming: one server, one client
- The server translates domain names to IP addresses
  - It has a small domain name to IP address map with five entries
  - The map can be coded that it resides in the memory of the server
  - It receives domain name query messages from the client and replies its corresponding IP addresses
  - Example: python3 server.py
- The client sends the domain name query from the command line
  - Example: python3 client.py
  - The message format follows the DNS message format discussed in the lecture notes. Please see detailed description in the lab manual

# DNS Application – Terminal Output

- Client

**Input from the user:**

> Enter Domain Name: google.com

**Output:**

> google.com: type A, class IN, TTL 260, addr (4) 192.165.1.1

> google.com: type A, class IN, TTL 260, addr (4) 192.165.1.10

represents the length of the records in bytes

- Server

**Request:**

ff 9b 81 80 00 01 00 02 00 00 00 00 07 68 6f 77
63 6f 64 65 03 6f 72 67 00 00 01 00 01

**Response:**

ff 9b 81 80 00 01 00 02 00 00 00 00 07 68 6f 77
63 6f 64 65 03 6f 72 67 00 00 01 00 01 c0 0c 00
01 00 01 00 00 01 26 00 04 68 1c 0e 71 c0 0c 00
01 00 01 00 00 01 26 00 04 68 1c 0f 71

# Submissions – Task1 Web Server

- Code
  - webserver.py code
  - README with instructions on how to execute the code
- Report
  - Explain your code
  - Screenshots of our client browser, verifying that the correct .html file is received

# Submissions – Task 2 DNS Client-Server

- Code
  - The server.py and client.py
  - README with instructions on how to execute the code
- Report
  - Explain your code
  - Screenshot of the output when client enters “google.ca”
  - Color code the DNS query message (in Hex)
  - Color code the DNS response message (in Hex)

1a 2b 04 00 00 01 00 00 00 00 00 00 06 67 6f 6f

67 6c 65 02 63 61 00 00 01 00 01

| Type | Key | Value |
|---|---|---|
| DNS HEADER | ID | 1a 2b |
| | FLAGS | 04 00 |
| | QDCOUNT | 00 01 |
| | ANCOUNT | 00 00 |
| | NSCOUNT | 00 00 |
| | ARCOUNT | 00 00 |
| QUERY | QNAME | 06 67 6f 6f 67 6c 65 02 63 61 00 |
| | QTYPE | 00 01 |
| | QCLASS | 00 01 |

# Submission – Task 3 and Final Report

- Task 3: Answer all the pencil-paper questions
- Final Report
  - Table of contents
  - Answer all the questions and provide explanations as asked for
  - Source code with proper documentation/comments
    - README with clear instruction on how to execute the code

You might be asked to demo your code

# References

[1] J. F. Kurose and K. W. Ross: *Computer Networking, A Top-Down Approach.* 8th Edition.

[2] P. Mockapetris, Domain Names – Implementation and Specification, document RFC 1035, 1987.