

B. Steenis – Networks 2

Session 5 : Sockets Python Lab - **Warning : there is a mandatory report to produce about this lab.**

1. Python
2. Sockets
3. Simple application sockets with python + wireshark

Why Python ?

Python is a [high-level](#), [general-purpose programming language](#). Its design philosophy emphasizes [code readability](#) with the use of [significant indentation](#).

Python is [dynamically typed](#) and [garbage-collected](#). It supports multiple [programming paradigms](#), including [structured](#) (particularly [procedural](#)), [object-oriented](#) and [functional programming](#). It is often described as a "batteries included" language due to its comprehensive [standard library](#).

(wikipedia)

Advantages

- Python is easy to use, easy to learn. Extremely fast learning curve. Extensive help available online.
- Source code is extremely readable, due to indentation implementing programming structures like loops or tests
- Requires no IDE (Integrated development environment)
- Usually interpreted (not compiled) but compilers do exist
- Source code can be view as a script. Can be executed line per line in commad line mode
- Huge community of developers
- Hundreds of available libraries covering almlost everything
- Old langage (first appeared 1991, but recently became popular)
- Mostly used in academic and education world, and for development/testing

The only disadvantage is that execution is not as fast as compiled C/C++ ,
and if you try to compile, Python compilers are generally not efficient (producing huge executable files)

Windows : Install from website : python.org

Linux : already preinstalled on most linux distributions.

Python is command-line. **So, you have to add the directory where you installed it in the system PATH environment.**

← → ↻

https://www.python.org

☆

🔒

📶

ABP

Python


PSF

Docs


PyPI

Jobs

Community



Donate



GO

Socialize

About

Downloads

Documentation

Community

Success Stories

News

Events

```
# For loop on a list
>>> numbers = [2, 4, 6, 8]
>>> product = 1
>>> for number in numbers:
...     product = product * number
...
>>> print('The product is:', product)
The product is: 384
```

>_

All the Flow You'd Expect

Python knows the usual control flow statements that other languages speak — `if`, `for`, `while` and `range` — with some of its own twists, of course. [More control flow tools in Python 3](#)

1


2


3


4


5

Python is a programming language that lets you work quickly and integrate systems more effectively. [>>> Learn More](#)

 **Get Started**
Whether you're new to programming or an experienced developer, it's easy to learn and use Python.
[Start with our Beginner's Guide](#)

 **Download**
Python source code and installers are available for download for all versions!
Latest: [Python 3.11.2](#)

 **Docs**
Documentation for Python's standard library, along with tutorials and guides, are available online.
[docs.python.org](#)

 **Jobs**
Looking for work or have a Python related position that you're trying to hire for? Our **relaunched community-run job board** is the place to go.
[jobs.python.org](#)

First test ...

On Windows prompt, type

```
> python
```

On Linux prompt, type

```
$ python3
```

Python prompt is

```
>>>
```

On python prompt, you can type any python code. It is immediately executed ...

```
>>> print ("Hello world")
```

There is no termination character like « ; »

You can also type a loop structure

```
>>> for i in range(10): print(i)
```

Please remark there is a « : » after the for loop statement. It is the same after a test statement

Second remark, after typing this line you get another prompt, this time is triple point

```
...
```

This prompt indicates the statement is incomplete.

If you typed something the same line then the for/while/if ... then you must type a blank line (directly with a carriage return), which is the case here

Otherwise you can type the content of the loop or test structure USING INDENTATION

Example :

```
>>> for i in range(2,4):
```

```
...     print(i)
```

```
...     print(2*i)
```

Then always terminate with a blank line

Example of a test structure

```
>>> i=1
```

```
>>> if i==1:
```

```
...     print("Equal to 1")
```

```
... else:
```

```
...     print("Not equal to 1")
```

Remark the test structure, and the indentation. You can of course type multiple lines between if and else and else and the final line

To exit python prompt type

```
>>> quit()
```

```
bsteenis@LAPTOP-9K4HD07Q: ~  
bsteenis@LAPTOP-9K4HD07Q:~$ python3  
Python 3.8.10 (default, Jun  2 2021, 10:49:15)  
[GCC 9.4.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("Hello world")  
Hello world  
>>> for i in range(10): print(i)  
...  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
>>> for i in range(2,4):  
...     print (i)  
...     print (2*i)  
...  
2  
4  
3  
6  
>>> i=1  
>>> if i==1 :  
...     print("Equal")  
... else:  
...     print("Not equal")  
...  
Equal  
>>> quit()  
bsteenis@LAPTOP-9K4HD07Q:~$
```

This is an example on a Linux shell. Same in Windows shell.

Of course the source code can be in a text file (source file). You can edit the source with your preferred text editor like Notepad in Windows ...

Insert this simple code in a text file named, for example, « eratos.py »

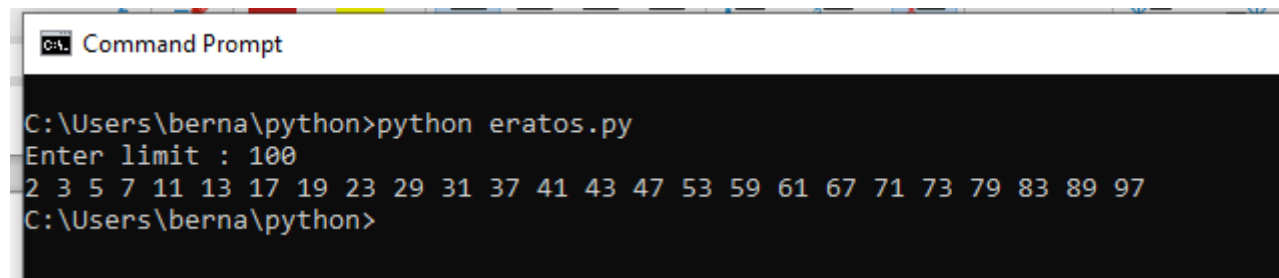
Note that usual extension for python sources is py ...

```
#test python : sieve of Eratosthenes
import math
lim=input("Enter limit : ")
lim=int(lim)+1
a=[]
for i in range(lim): a.append(1)
a[1]=0
for i in range(2,1+int(math.sqrt(lim))):
    if a[i]==1:
        for j in range(2*i,lim,i):
            a[j]=0
for i in range(1,lim):
    if a[i]==1: print(i,end=" ")
```

Then on Windos or Linux prompt, just type

> **python eratos.py**

to execute this source ...



```
C:\Users\berna\python>python eratos.py
Enter limit : 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
C:\Users\berna\python>
```

This is Eratosthene's algorithm to get the list of prime numbers to a given limit. The interest of this small piece of code is that it gives a extensive viewpoint of all programing features ...

- Against, be careful about indentation. This is really a key of Python sources.

- # is a comment

- import is a library declaration. All libraries should be installed. « math » library is present by default. It is requested for square root sqrt() function

- no variable declaration, they are implicitly declared at first assignation

- input() function to get input from stdin (here input prompt)

- int() is a conversion function to integer

- look at array or structure construction ... [] is void array, then append method is applied to add elements

- for / range loop and tests

There is a lot of help available online. Never hesitate to type python and a feature/function/command in search engine, like « python while loop »
For core python : Official help from **docs.python.org**

The screenshot shows a web browser window displaying the Python Standard Library documentation for version 3.11.2. The browser's address bar shows the URL `https://docs.python.org/3.11/library/index.html`. The page header includes the Python logo, a language dropdown set to 'English', a version dropdown set to '3.11.2', and a breadcrumb trail: '3.11.2 Documentation » The Python Standard Library'.

Left Sidebar:

- Previous topic:** 10. Full Grammar specification
- Next topic:** Introduction
- This Page:**
 - [Report a Bug](#)
 - [Show Source](#)

Main Content:

The Python Standard Library

While [The Python Language Reference](#) describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

Python's standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

In addition to the standard library, there is an active collection of hundreds of thousands of components (from individual programs and modules to packages and entire application development frameworks), available from the [Python Package Index](#).

- [Introduction](#)
 - [Notes on availability](#)
- [Built-in Functions](#)
- [Built-in Constants](#)
 - [Constants added by the site module](#)
- [Built-in Types](#)
 - [Truth Value Testing](#)
 - [Boolean Operations — and, or, not](#)
 - [Comparisons](#)
 - [Numeric Types — int, float, complex](#)
 - [Iterator Types](#)
 - [Sequence Types — list, tuple, range](#)
 - [Text Sequence Type — str](#)

An alternative interesting help source is on www.w3schools.com/python/

The screenshot shows the W3Schools Python Tutorial page. The browser's address bar shows the URL www.w3schools.com/python/. The navigation menu at the top includes links for HTML, CSS, JAVASCRIPT, SQL, PYTHON (highlighted), JAVA, PHP, BOOTSTRAP, HOW TO, W3.CSS, and C. The left sidebar contains a list of Python topics, with 'Python Arrays' highlighted. The main content area features a yellow note at the top stating: 'Note: The list's `remove()` method only removes the first occurrence of the specified value.' Below this is the section 'Array Methods', which explains that Python has a set of built-in methods for lists/arrays. A table lists 12 methods: `append()`, `clear()`, `copy()`, `count()`, `extend()`, `index()`, `insert()`, `pop()`, `remove()`, `reverse()`, and `sort()`. A final yellow note at the bottom states: 'Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.'

Python Tutorial

- Python HOME
- Python Intro
- Python Get Started
- Python Syntax
- Python Comments
- Python Variables
- Python Data Types
- Python Numbers
- Python Casting
- Python Strings
- Python Booleans
- Python Operators
- Python Lists
- Python Tuples
- Python Sets
- Python Dictionaries
- Python If...Else
- Python While Loops
- Python For Loops
- Python Functions
- Python Lambda
- Python Arrays**
- Python Classes/Objects
- Python Inheritance
- Python Iterators
- Python Scope
- Python Modules
- Python Dates
- Python Math
- Python JSON
- Python RegEx
- Python PIP
- Python Try...Except

Note: The list's `remove()` method only removes the first occurrence of the specified value.

Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
<code>append()</code>	Adds an element at the end of the list
<code>clear()</code>	Removes all the elements from the list
<code>copy()</code>	Returns a copy of the list
<code>count()</code>	Returns the number of elements with the specified value
<code>extend()</code>	Add the elements of a list (or any iterable), to the end of the current list
<code>index()</code>	Returns the index of the first element with the specified value
<code>insert()</code>	Adds an element at the specified position
<code>pop()</code>	Removes the element at the specified position
<code>remove()</code>	Removes the first item with the specified value
<code>reverse()</code>	Reverses the order of the list
<code>sort()</code>	Sorts the list

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.


Or ... www.geeksforgeeks.org/python-programming-language/

The screenshot shows a web browser displaying the GeeksforGeeks website. The address bar shows the URL: <https://www.geeksforgeeks.org/gfact-50-python-end-parameter-in-print/?ref=lbj>. The website has a navigation bar with links to Courses, Tutorials, Jobs, Practice, and Contests. Below this is a secondary navigation bar with links to DSA, Data Structures, Algorithms, Interview Preparation, Data Science, Topic-wise Practice, C, C++, Java, Python, and Latest Blogs. A sidebar on the left contains links to 'Write an Article', 'Write an Interview Experience', 'Introduction', and 'Input/Output'. Under 'Input/Output', there are links to 'Taking input in Python', 'Taking input from console in Python', 'Taking multiple inputs from user in Python', 'Python | Output using print() function', 'How to print without newline in Python?', 'Python end parameter in print()', and 'Python | sep parameter in print()'. The main content area features the article title 'Python end parameter in print()' with a difficulty level of 'Easy' and a last update date of '23 Feb, 2023'. Below the title are tabs for 'Read', 'Discuss', 'Courses', 'Practice', and 'Video'. The article text explains that by default, Python's print() function ends with a newline, and the 'end' parameter can be used to change this. An example shows how to use the 'end' parameter to print two lines without a newline. The right sidebar shows a 'Courses' section with a recommendation for 'Data Structures & Algorithms in Python - Sel...'.

File Edit View History Bookmarks Tools Help

W Gigab W Line c W 6b/8b W Duple Welco W Pytho G pytho W Pytho Pytho Welco G pytho ee Pyt X

← → ↻ <https://www.geeksforgeeks.org/gfact-50-python-end-parameter-in-print/?ref=lbj> ☆

Courses ▾ Tutorials ▾ Jobs ▾ Practice ▾ Contests ▾  🔍 🌙 📱

← DSA Data Structures Algorithms Interview Preparation Data Science Topic-wise Practice C C++ Java Python Latest Blogs

Write an Article ▾

Write an Interview Experience

Introduction ▾

Input/Output ^

Taking input in Python

Taking input from console in Python

Taking multiple inputs from user in Python

Python | Output using print() function

How to print without newline in Python?

Python end parameter in print()

Python | sep parameter in print()

Python end parameter in print()

Difficulty Level : Easy • Last Updated : 23 Feb, 2023

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

By default [Python's](#) print() function ends with a newline. A programmer with C/C++ background may wonder how to print without a newline. [Python's print\(\)](#) function comes with a parameter called 'end'. By default, the value of this parameter is '\n', i.e. the new line character.


Example 1:

Here, we can end a print statement with any character/string using this parameter.

Python3

```
# ends the output with a space
print("Welcome to", end = ' ')
print("GeeksforGeeks", end= ' ')
```

Courses

 111k+ interested Geeks

Data Structures & Algorithms in Python - Sel...

Beginner to Advance

AD

This list of help/tutorial websites is of course not complete. There are really a lot.
For socket programming we'll have to use the « socket » library which is standard in python.
For external libraries, developers often provide their own online help pages.

Python » English » 3.11.2 » 3.11.2 Documentation » The Python Standard Library » Networking and Interprocess Communication » **socket** — Low-level networking interface

Table of Contents

- socket** — Low-level networking interface
 - Socket families
 - Module contents
 - Exceptions
 - Constants
 - Functions
 - Creating sockets
 - Other functions
 - Socket Objects
 - Notes on socket timeouts
 - Timeouts and the **connect** method
 - Timeouts and the **accept** method
 - Example

Previous topic

Developing with asyncio

Next topic

ssl — TLS/SSL wrapper for socket objects

This Page

Report a Bug

Show Source

socket — Low-level networking interface

Source code: [Lib/socket.py](#)

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

Note: Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

Availability: not Emscripten, not WASI.

This module does not work or is not available on WebAssembly platforms `wasm32-emscripten` and `wasm32-wasi`. See [WebAssembly platforms](#) for more information.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the `socket()` function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

See also:

Module [socketserver](#)

Classes that simplify writing network servers.

Module [ssl](#)

A TLS/SSL wrapper for socket objects.

Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

About the socket library, one important issue is that the behaviour is strongly platform dependent, and rely on OS APIs
In the case of Windows, only TCP and UDP transport protocols are accepted, and it seems (almost) impossible to access directly to the IP network layer or even below to the Link layer without going through the transport layer.

Otherwise to conclude the huge list of available online help with Python, forums can sometimes be very helpful. If you face a big problem and don't see the solution, think that someone already faced the same problem before you ...

The **stackoverflow.com** forum is a gold mine.

The screenshot shows a web browser displaying a Stack Overflow question. The URL in the address bar is <https://stackoverflow.com/questions/65389606/how-to-handle-both-ipv4-and-ipv6-client-request>. The page header includes the Stack Overflow logo, navigation links (About, Products, For Teams), a search bar, and buttons for 'Log in' and 'Sign up'. A blue banner below the header asks for a survey. The left sidebar contains navigation links: Home, PUBLIC, Questions (selected), Tags, Users, Companies, COLLECTIVES (with an 'Explore Collectives' link), and TEAMS (with a 'Stack Overflow for Teams' section and a 'Create a free Team' button). The main content area features the question title 'How to handle both ipv4 and ipv6 client request' with an 'Ask Question' button. Below the title, it says 'Asked 2 years, 3 months ago', 'Modified 2 years, 2 months ago', and 'Viewed 3k times'. The question body starts with 'I am new to socket programming, My use case is to handle client request for both ipv4 and ipv6. Right now ipv4 is working fine but not able to use ipv6 Ip.' followed by a code snippet: '// Run Server Python server.py -i " -p 2010'. A note explains the double quote: 'Note: " Means I want to listen to the client at both addresses.' Below this is a code block for 'server.py' showing Python socket code. To the right of the code block is a 'client.py' code block. On the far right, there is a 'The Overflow Blog' section with two articles and a 'Featured on Meta' section with five items.

Home
PUBLIC
Questions
Tags
Users
Companies
COLLECTIVES
Explore Collectives
TEAMS
Stack Overflow for Teams – Start collaborating and sharing organizational knowledge.
Free
Create a free Team

How to handle both ipv4 and ipv6 client request

Asked 2 years, 3 months ago Modified 2 years, 2 months ago Viewed 3k times

1

I am new to socket programming, My use case is to handle client request for both ipv4 and ipv6. Right now ipv4 is working fine but not able to use ipv6 Ip.

// Run Server Python server.py -i " -p 2010

Note: " Means I want to listen to the client at both addresses.

```
server.py

tcpServer = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcpServer.bind((self.TCP_IP, self.TCP_Port))
tcpServer.listen(4)

while True:
    print "Multithreaded Python server : Waiting for connection"
    (conn, (ip, port)) = tcpServer.accept()
    print("Client connected: {}".format(ip, port))
```

client.py

```
host = ":::1"
port = 2010
```

The Overflow Blog

- Who builds it and who runs it? SRE team topologies
- What our engineers learned building Stack Overflow (Ep. 547)

Featured on Meta

- We've added a "Necessary cookies only" option to the cookie consent popup
- The Stack Exchange reputation system: What's working? What's not?
- Should we burninate [rowname] and [columnname]?
- Launching the CI/CD and R Collectives and community editing features for...
- Staging Ground Beta 1 Recap, and Reviewers needed for Beta 2
- Temporary policy: ChatGPT is banned

Application Layer

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System: DNS
- P2P applications
- video streaming, CDNs
- socket programming with UDP and TCP

COMPSCI 453 **Computer Networks**

Professor Jim Kurose

College of Information and Computer Sciences

University of Massachusetts



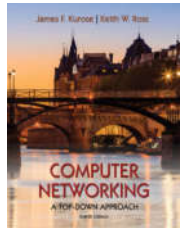
Class textbook:

Computer Networking: A Top-Down Approach (8th ed.)

J.F. Kurose, K.W. Ross

Pearson, 2020

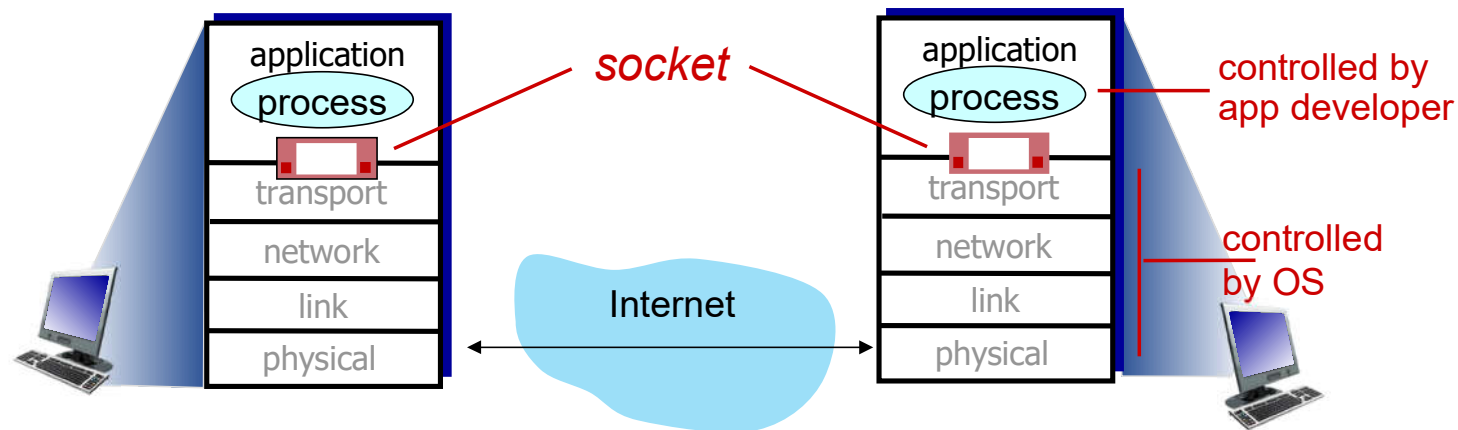
http://gaia.cs.umass.edu/kurose_ross



Socket programming

goal: learn how to build client/server applications that communicate using sockets

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



Socket programming

Two socket types for two transport services:

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

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 UDP

UDP: no “connection” between client and server:

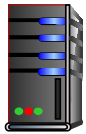
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes (“datagrams”) between client and server processes

Client/server socket interaction: UDP



server (running on serverIP)

create socket, port= x:
`serverSocket =
socket(AF_INET,SOCK_DGRAM)`

read datagram from
`serverSocket`

write reply to
`serverSocket`
specifying
client address,
port number

client

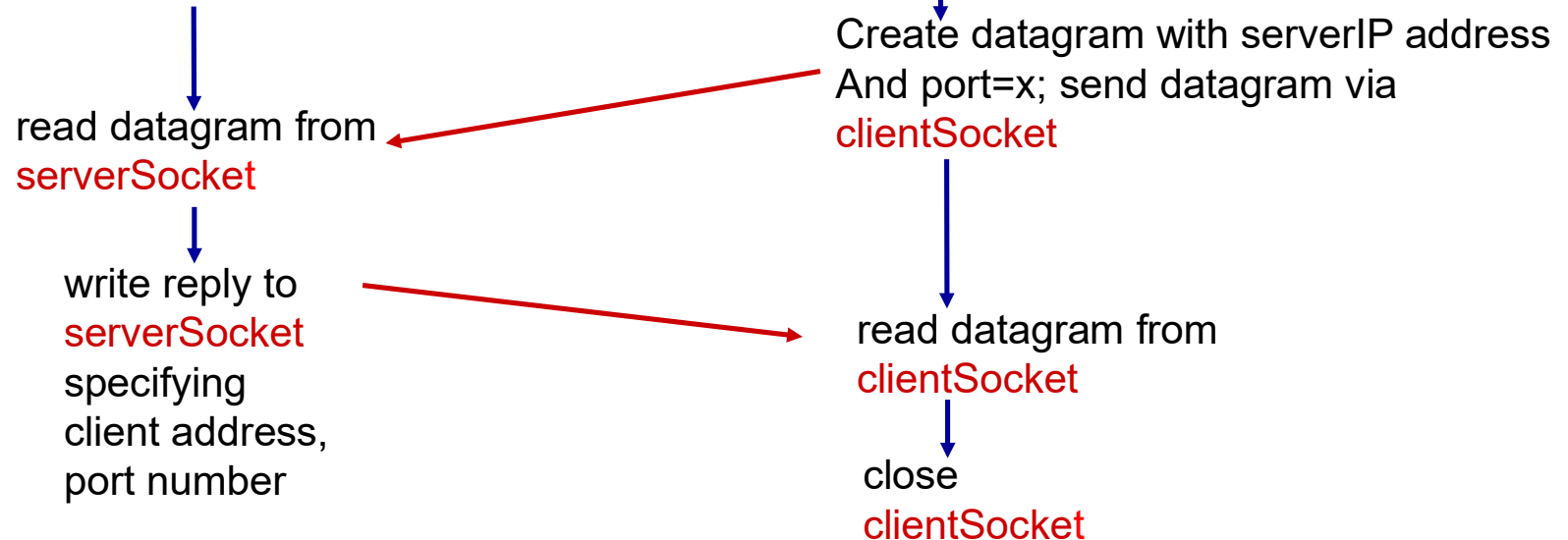


create socket:
`clientSocket =
socket(AF_INET,SOCK_DGRAM)`

Create datagram with serverIP address
And port=x; send datagram via
`clientSocket`

read datagram from
`clientSocket`

close
`clientSocket`



Example app: UDP client

Python UDPClient

include Python's socket library	→	from socket import *
		serverName = 'hostname'
		serverPort = 12000
create UDP socket for server	→	clientSocket = socket(AF_INET, SOCK_DGRAM)
get user keyboard input	→	message = raw_input('Input lowercase sentence:')
attach server name, port to message; send into socket	→	clientSocket.sendto(message.encode(), (serverName, 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()

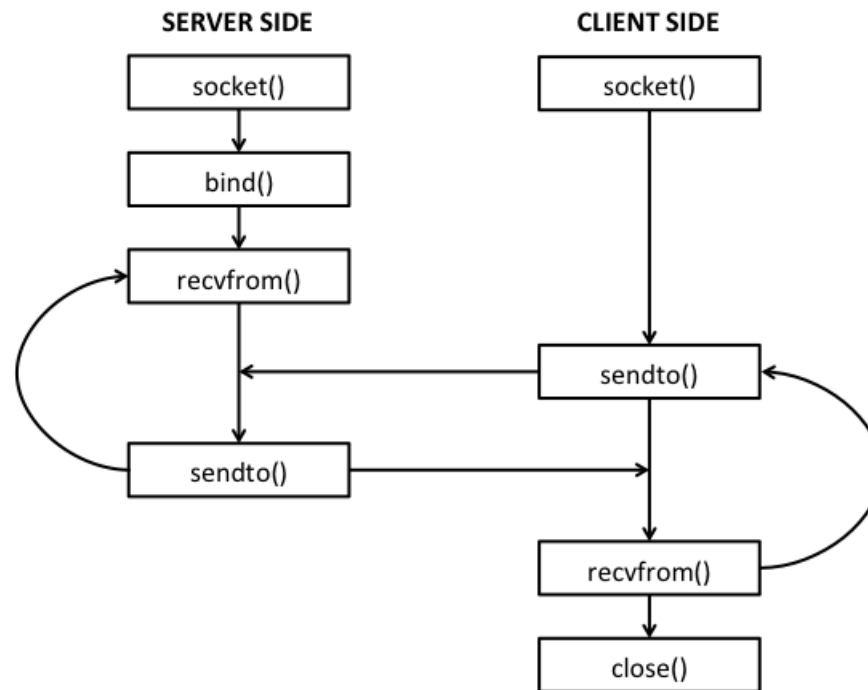
Example app: UDP server

Python UDPServer

	from socket import *
	serverPort = 12000
create UDP socket	→ serverSocket = socket(AF_INET, SOCK_DGRAM)
bind socket to local port number 12000	→ serverSocket.bind(("", serverPort))
	print ("The server is ready to receive")
loop forever	→ while True:
Read from UDP socket into message, getting client's address (client IP and port)	→ message, clientAddress = serverSocket.recvfrom(2048)
send upper case string back to this client	→ modifiedMessage = message.decode().upper() serverSocket.sendto(modifiedMessage.encode(), clientAddress)

```
# place this code in file client_udp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))    # socket server port number
server = (host,port)
client_socket = socket.socket(family=socket.AF_INET,type=socket.SOCK_DGRAM)
message = input(" -> ")  # take input
while message.lower().strip() != 'bye':
    client_socket.sendto(message.encode(),server)
    data,addr = client_socket.recvfrom(1024)
    data=data.decode()  # receive response
    print('Received from server: ')
    print(data)  # show in terminal
    message = input(" -> ")  # again take input
client_socket.close()  # close the connection
```

```
# place this code in file server_udp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket(family=socket.AF_INET,type=socket.SOCK_DGRAM)
s.bind((host, port))
print("Server Started")
while True:
    data, address = s.recvfrom(1024)
    data = data.decode('utf-8')
    if not data: break
    print("Message from: " + str(address))
    print("Received : ")
    print(data)
    data = data.upper()
    print("Sending: ")
    print(data)
    s.sendto(data.encode('utf-8'),address)
s.close()
```



(<https://www.it.uu.se/education/course/homepage/dsp/vt19/modules/module-2/sockets/>)

socket.socket (family=socket.AF_INET, type=socket.SOCK_DGRAM)

« socket dot something » is because the function or the constant is defined in the socket library.

`socket()` creates a socket with two main parameters :

family is the IP protocol, `AF_INET` for IPv4, `AF_INET6` for IPv6. `AF_INET` (IPv4) is by default and can be omitted

type is the Transport protocol, `SOCK_DGRAM` for UDP, `SOCK_STREAM` for TCP. `SOCK_STREAM` (TCP) is by default and can be omitted.

There are other parameters and many other options defined in the online documentation, however only these combinations work in Windows API.

`socket()` creates the socket with the IP address of the machine and a randomly generated port. This is convenient for client side.

`bind(address,port)` makes association between the socket and a given address and port. It is necessary to specify at least a listening port on server side. Address can be left blank, in that case server is listening on all interfaces, or if an address is specified, then server is only listening to the associated interface.

`sendto(address,port)` sends a packet to the specified destination (address and port)

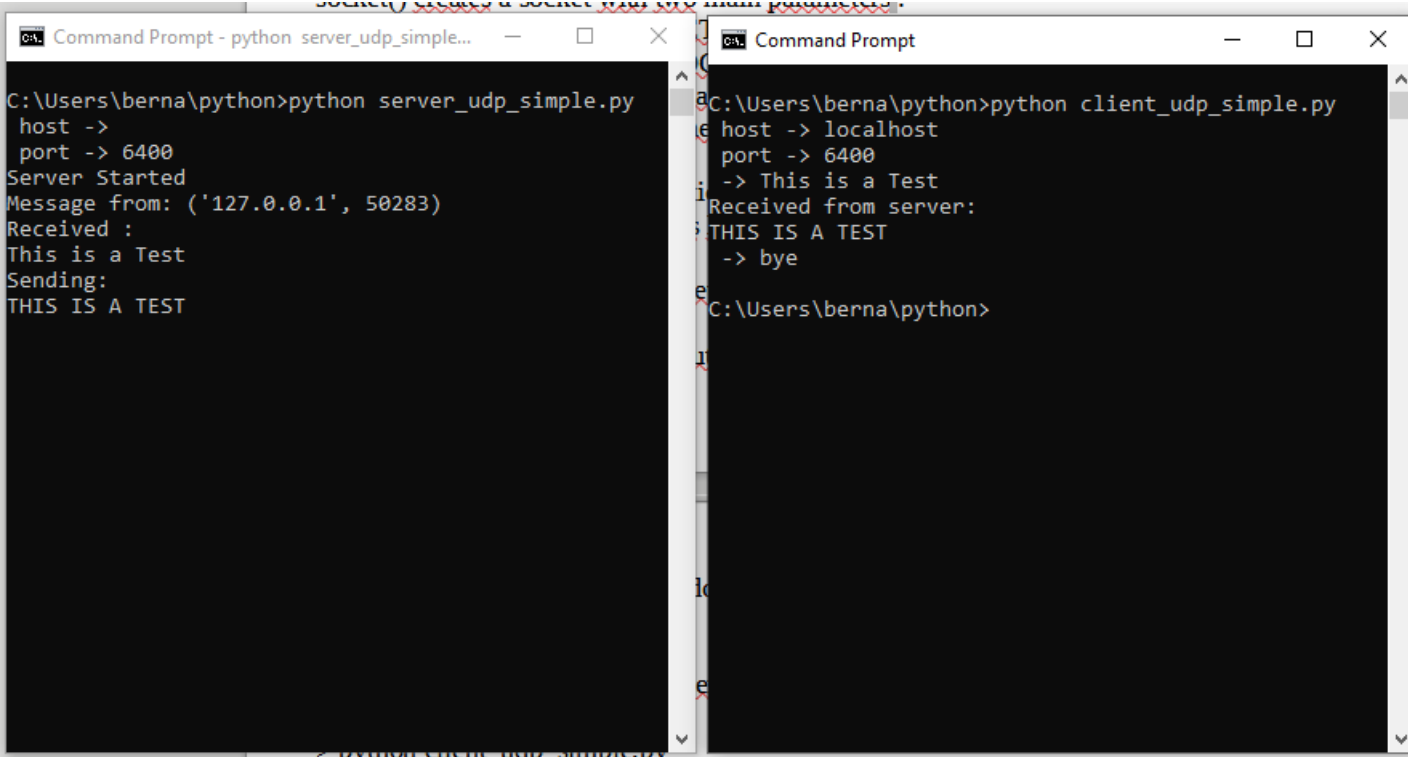
`recvfrom()` waits and blocks execution until reception of a packet, data and sending address are both returned by the function. Parameter is buffer size.

Open two command shell in Windows, and launch in the first shell
> python server_udp_simple.py
On host input, leave blank
On port input type any port number you wish, for example 6000
In the second shell, launch
> python client_udp_simple.py
On host input, type localhost
On port input, type the same port number as in the server program.

Please remark that an address (host) may be a name, not necessary an IP address. Then the Windows API will perform the appropriate DNS translation.

On prompt in the client program, type any text you wish, it will be transmitted to the server. Upon reception, the server will upper case all characters and send them back to the client. That is all that our server does ... echoing in uppercase (this is just how it is coded inn the server, and of course this is just for testing:)

If you type « bye » in the client window, it will close the connection on its side (this is just how it is coded in the client). Note that the server will not close anything and will continue running
(you can press CTRL-Break to stop the process)

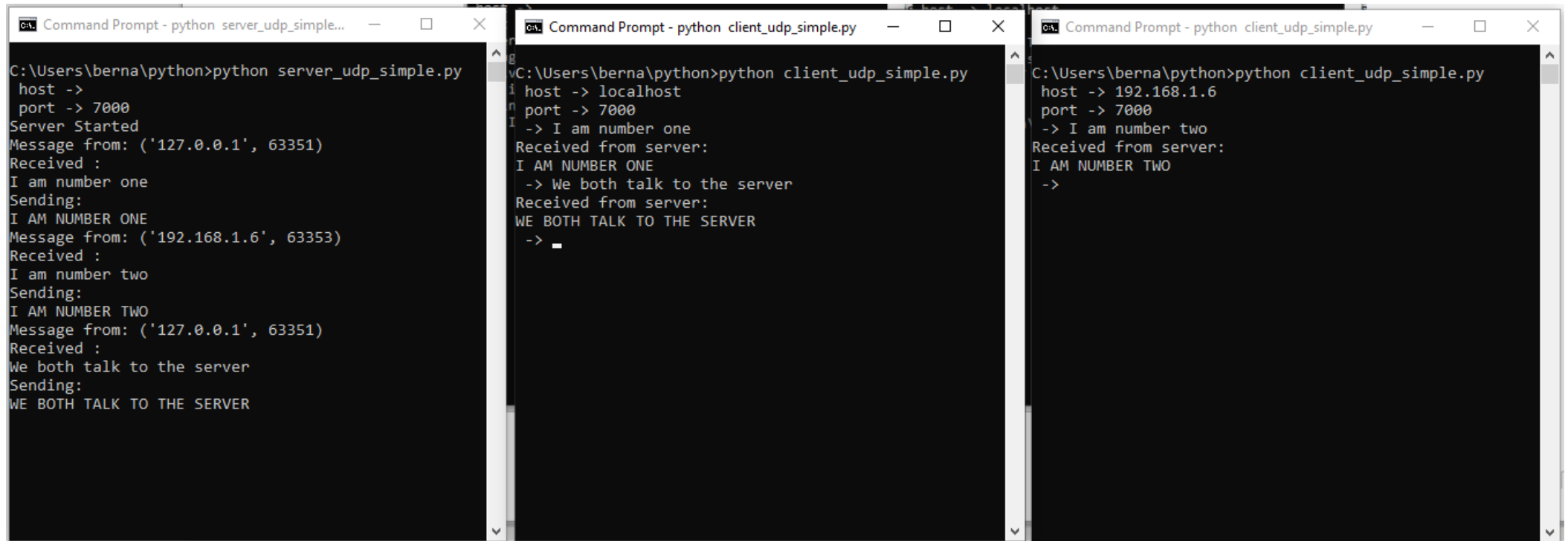


```
C:\Users\berna\python>python server_udp_simple.py
host ->
port -> 6400
Server Started
Message from: ('127.0.0.1', 50283)
Received :
This is a Test
Sending:
THIS IS A TEST

C:\Users\berna\python>python client_udp_simple.py
host -> localhost
port -> 6400
-> This is a Test
Received from server:
THIS IS A TEST
-> bye

C:\Users\berna\python>
```

Please note that you can open as many clients as you want and send messages to the server ...



```
Command Prompt - python server_udp_simple.py
C:\Users\berna\python>python server_udp_simple.py
host ->
port -> 7000
Server Started
Message from: ('127.0.0.1', 63351)
Received :
I am number one
Sending:
I AM NUMBER ONE
Message from: ('192.168.1.6', 63353)
Received :
I am number two
Sending:
I AM NUMBER TWO
Message from: ('127.0.0.1', 63351)
Received :
We both talk to the server
Sending:
WE BOTH TALK TO THE SERVER

Command Prompt - python client_udp_simple.py
C:\Users\berna\python>python client_udp_simple.py
host -> localhost
port -> 7000
-> I am number one
Received from server:
I AM NUMBER ONE
-> We both talk to the server
Received from server:
WE BOTH TALK TO THE SERVER
->

Command Prompt - python client_udp_simple.py
C:\Users\berna\python>python client_udp_simple.py
host -> 192.168.1.6
port -> 7000
-> I am number two
Received from server:
I AM NUMBER TWO
->
```

Please note also that the server accepts messages from all the clients as the notion of connection does not exist, each datagram/packet is an individual message by itself and all the datagrams are independent from each other.

Please also note that first client has established a connection to « localhost » which is translated to IPv4 loopback address 127.0.0.1, while the second client established a connection to 192.168.1.6 which is the present IPv4 address of my laptop. Of course you should check your IP with ipconfig and type the correct one otherwise the connection will be denied, and as there is no error processing routine in those simple Python codes, the client program will probably crash if there is no server to answer the request.

If I specified localhost on the server host prompt, then the client connecting to localhost would be accepted while the client connecting to 192.168.1.6 would be denied.

We have seen that IPv4 is by default. We can modify a little bit the codes to implement a choice between the two IP versions :

```

# place this code in file client_udp_ipchoice.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))    # socket server port number
ipversion = input(" use ipv6 ? type y for yes -> ").lower()
ipv = socket.AF_INET
if ipversion=='y': ipv=socket.AF_INET6
server = (host,port)
client_socket = socket.socket(family=ipv,type=socket.SOCK_DGRAM)
message = input(" -> ")  # take input
while message.lower().strip() != 'bye':
    client_socket.sendto(message.encode(),server)
    data,addr = client_socket.recvfrom(1024)
    data=data.decode()  # receive response
    print('Received from server: ')
    print(data)  # show in terminal
    message = input(" -> ")  # again take input
client_socket.close()  # close the connection

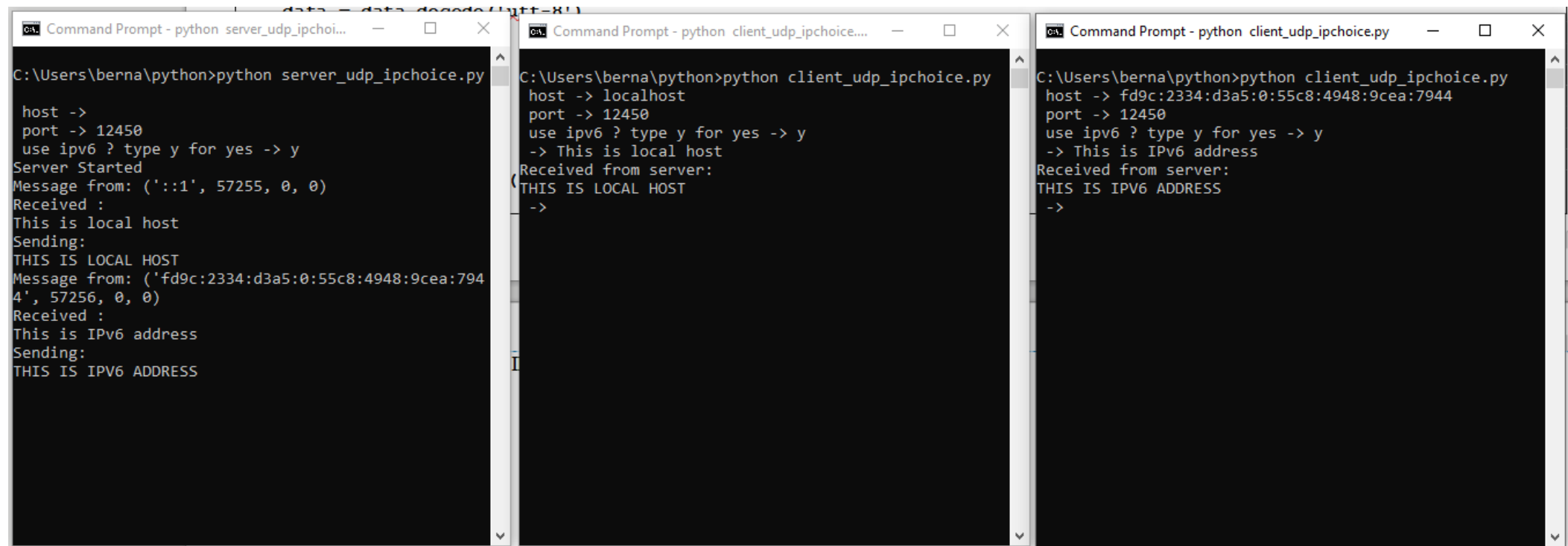
```

```

# place this code in file server_udp_ipchoice.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
ipversion = input(" use ipv6 ? type y for yes -> ").lower()
ipv = socket.AF_INET
if ipversion=='y': ipv=socket.AF_INET6
s = socket.socket(family=ipv,type=socket.SOCK_DGRAM)
s.bind((host, port))
print("Server Started")
while True:
    data, address = s.recvfrom(1024)
    data = data.decode('utf-8')
    if not data: break
    print("Message from: " + str(address))
    print("Received : ")
    print(data)
    data = data.upper()
    print("Sending: ")
    print(data)
    s.sendto(data.encode('utf-8'),address)
s.close()

```

So now we can try the same with IPv6 ...



```
Command Prompt - python server_udp_ipchoi...
C:\Users\berna\python>python server_udp_ipchoice.py
host ->
port -> 12450
use ipv6 ? type y for yes -> y
Server Started
Message from: ('::1', 57255, 0, 0)
Received :
This is local host
Sending:
THIS IS LOCAL HOST
Message from: ('fd9c:2334:d3a5:0:55c8:4948:9cea:7944', 57256, 0, 0)
Received :
This is IPv6 address
Sending:
THIS IS IPV6 ADDRESS

Command Prompt - python client_udp_ipchoice...
C:\Users\berna\python>python client_udp_ipchoice.py
host -> localhost
port -> 12450
use ipv6 ? type y for yes -> y
-> This is local host
Received from server:
THIS IS LOCAL HOST
->

Command Prompt - python client_udp_ipchoice.py
C:\Users\berna\python>python client_udp_ipchoice.py
host -> fd9c:2334:d3a5:0:55c8:4948:9cea:7944
port -> 12450
use ipv6 ? type y for yes -> y
-> This is IPv6 address
Received from server:
THIS IS IPV6 ADDRESS
->
```

Once again we see that the server accepts datagram either from localhost (translated to ::1) as from the real IPv6 address. Of course client and server must use the same Ipversion otherwise connection will be denied by server and client program will crash.

But there should be a solution to implement a server that accepts both IPv4 and IPv6 clients ? No ? Solution found in forums ...

<https://stackoverflow.com/questions/65389606/how-to-handle-both-ipv4-and-ipv6-client-request>

The solution is to set correct socket options (using the setsockopt() command)

```
# place this code in file server_udp_ipboth.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket(family=socket.AF_INET6, type=socket.SOCK_DGRAM)
s.setsockopt(socket.IPPROTO_IPV6, socket.IPV6_V6ONLY, 0)
s.bind((host, port))
print("Server Started")
while True:
    data, address = s.recvfrom(1024)
    data = data.decode('utf-8')
    if not data: break
    print("Message from: " + str(address))
    print("Received : ")
    print(data)
    data = data.upper()
    print("Sending: ")
    print(data)
    s.sendto(data.encode('utf-8'), address)
s.close()
```

Of course this is not yet perfect, we normally have to check before implementing this if API supports dual stack protocol ...
In fact, what happen is that API will translate IPv4 address into an IPv6 to which the server will respond ... Here it works !!

```

C:\Users\berna\python>python server_udp_ipboth.py
 host -> 
 port -> 8000
Server Started
Message from: ('::1', 50547, 0, 0)
Received :
test v6
Sending:
TEST V6
Message from: ('::ffff:127.0.0.1', 50277, 0, 0)
Received :
test v4
Sending:
TEST V4

C:\Users\berna\python>python client_udp_ipchoice.py
 host -> localhost
 port -> 8000
 use ipv6 ? type y for yes -> y
 -> test v6
Received from server:
TEST V6
 -> bye

C:\Users\berna\python>

C:\Users\berna\python>python client_udp_ipchoice.py
 host -> localhost
 port -> 8000
 use ipv6 ? type y for yes -> n
 -> test v4
Received from server:
TEST V4
 -> bye

C:\Users\berna\python>

```

Now we will see the clear difference between UDP which is connectionless and TCP which establish a connection between client and server using the 3-way handshake protocol.

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 (more in Chap 3)

Application viewpoint

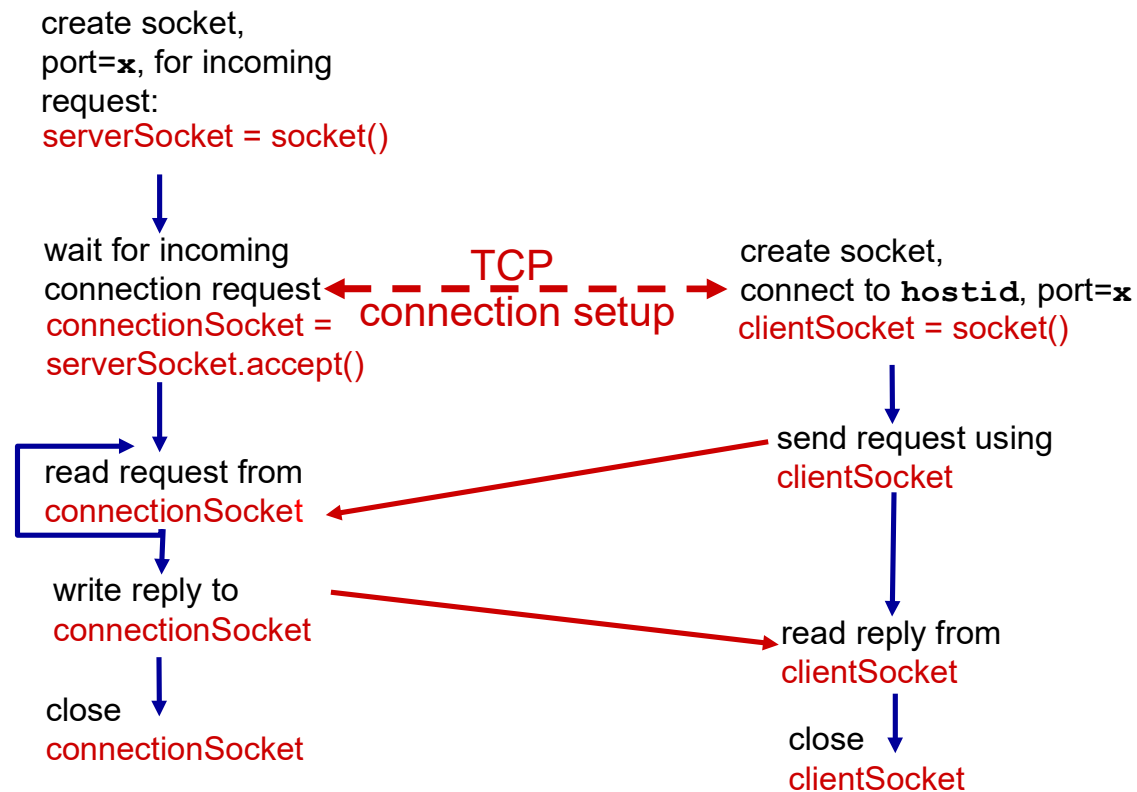
TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server processes

Client/server socket interaction: TCP



server (running on `hostid`)

client



Example app: TCP client

Python TCPClient

create TCP socket for server,
remote port 12000

```
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()
```

No need to attach server name, port

Example app: TCP server

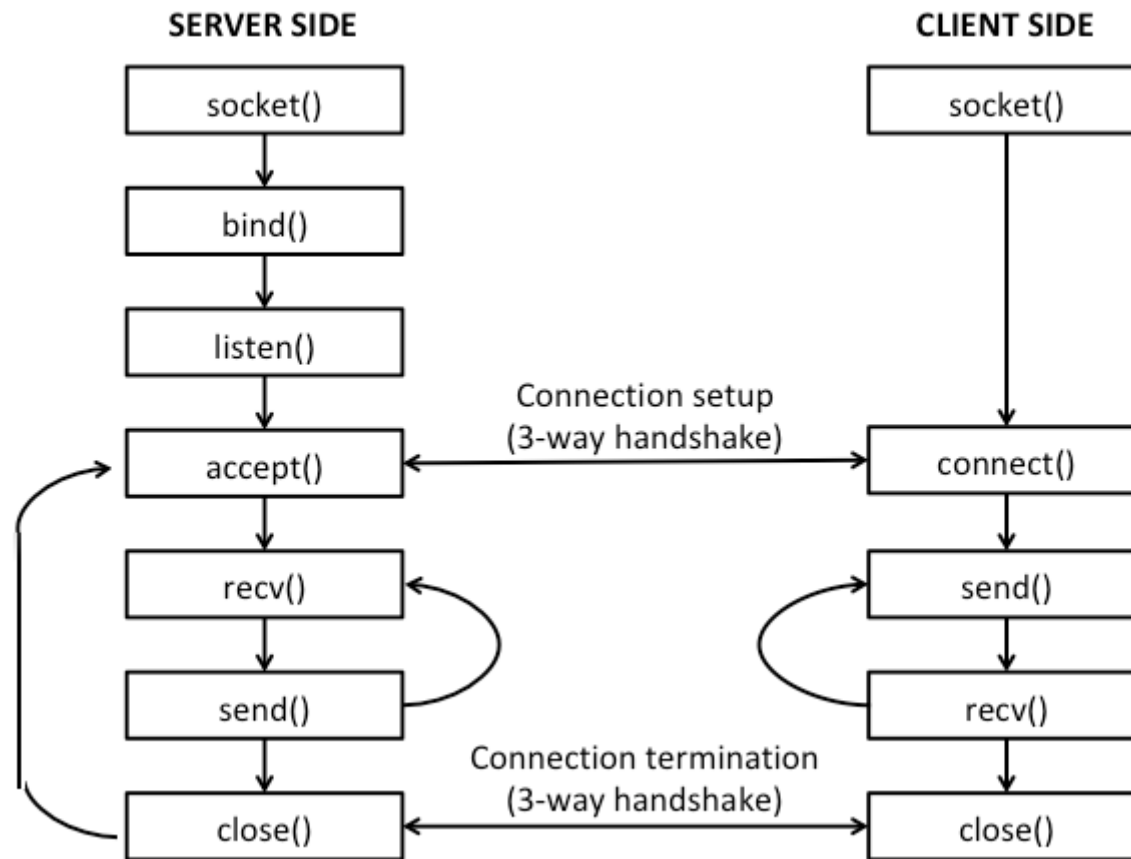
Python TCPServer

		<pre>from socket import *</pre>
		<pre>serverPort = 12000</pre>
create TCP welcoming socket	→	<pre>serverSocket = socket(AF_INET,SOCK_STREAM)</pre>
		<pre>serverSocket.bind(('',serverPort))</pre>
server begins listening for incoming TCP requests	→	<pre>serverSocket.listen(1)</pre>
		<pre>print 'The server is ready to receive'</pre>
loop forever	→	<pre>while True:</pre>
server waits on accept() for incoming requests, new socket created on return	→	<pre> connectionSocket, addr = serverSocket.accept()</pre>
		<pre> sentence = connectionSocket.recv(1024).decode()</pre>
read bytes from socket (but not address as in UDP)	→	<pre> capitalizedSentence = sentence.upper()</pre>
		<pre> connectionSocket.send(capitalizedSentence. encode())</pre>
close connection to this client (but <i>not</i> welcoming socket)	→	<pre> connectionSocket.close()</pre>

```
# place this code in file client_tcp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))    # socket server port number
server = (host,port)
client_socket = socket.socket(family=socket.AF_INET,type=socket.SOCK_STREAM)
client_socket.connect(server)    # connect to the server
message = input(" -> ")    # take input
while message.lower().strip() != 'bye':
    client_socket.send(message.encode())    # send message
    data,addr = client_socket.recvfrom(1024)
    data=data.decode()    # receive response
    print('Received from server: ')
    print(data)    # show in terminal
    message = input(" -> ")    # again take input
client_socket.close()    # close the connection
```

```
# place this code in file server_tcp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket()
s.bind((host, port))
print("Server Started")
s.listen(1)
while True:
    conn, address = s.accept()    # accept new connection
    print("Connection from: " + str(address))
    while True:
        data = conn.recv(1024).decode('utf-8')
        if not data: break
        print("Message from: " + str(address))
        print("Received : ")
        print(data)
        data = data.upper()
        print("Sending: ")
        print(data)
        conn.send(data.encode('utf-8'))
    conn.close()
    print("Connection closed")
s.close()
```

This is the schematic of TCP :



Let's first compare the two client codes : UDP and TCP :

```
# place this code in file client_udp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))    # socket server port number
server = (host,port)
client_socket = socket.socket(family=socket.AF_INET,type=socket.SOCK_DGRAM)
message = input(" -> ")  # take input
while message.lower().strip() != 'bye':
    client_socket.sendto(message.encode(),server)
    data,addr = client_socket.recvfrom(1024)
    data=data.decode()  # receive response
    print('Received from server: ')
    print(data)  # show in terminal
    message = input(" -> ")  # again take input
client_socket.close()  # close the connection
```

```
# place this code in file client_tcp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))    # socket server port number
server = (host,port)
client_socket = socket.socket(family=socket.AF_INET,type=socket.SOCK_STREAM)
client_socket.connect(server)  # connect to the server
message = input(" -> ")  # take input
while message.lower().strip() != 'bye':
    client_socket.send(message.encode())  # send message
    data,addr = client_socket.recvfrom(1024)
    data=data.decode()  # receive response
    print('Received from server: ')
    print(data)  # show in terminal
    message = input(" -> ")  # again take input
client_socket.close()  # close the connection
```

Let's first compare the two client codes : UDP and TCP :
 There is just one more line in TCP : the connect(server) command.

This initiates the TCP handshake.

The programmer doesn't have to take care about the 3-way handshake, because the Windows API does everything ...

Also note that the sento() comand doesn't specify the server anymore as it has been associated with the socket through the connect() command.

The server code is however completely different ...

```
# place this code in file server_tcp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket()
s.bind((host, port))
print("Server Started")
s.listen(1)
while True:
    conn, address = s.accept() # accept new connection
    print("Connection from: " + str(address))
    while True:
        data = conn.recv(1024).decode('utf-8')
        if not data: break
        print("Message from: " + str(address))
        print("Received : ")
        print(data)
        data = data.upper()
        print("Sending: ")
        print(data)
        conn.send(data.encode('utf-8'))
    conn.close()
    print("Connection closed")
s.close()
```

The bind() command that makes association between socket and its port number is immediately followed by a listen() command.

The parameter of listen() is arbitrary and is not necessary.

The accept() command is the counterpart to the connect() from the client. It establishes the connexion. But please note that connexion is requested and initiated by the client, the server is just waiting for a connection.

The accept() creates a new structure called a connection, that will replace the server for the receiving and sending commands.

The code contains two imbricated loops. Server will continue dialog with client until client close the connection (which is the not data test). Then server will come back to the outer loop waiting for a new connection.

Please note that as this code has no multithreading, if a second client try to connect while server is busy with first client, then this second client will be blocked until first client stops its connection.


```
Command Prompt - python server_tcp_simple...
C:\Users\berna\python>python server_tcp_simple.py
host ->
port -> 4400
Server Started
Connection from: ('127.0.0.1', 49760)
Message from: ('127.0.0.1', 49760)
Received :
This is client 1
Sending:
THIS IS CLIENT 1
Message from: ('127.0.0.1', 49760)
Received :
Client 2 is blocked until I terminate
Sending:
CLIENT 2 IS BLOCKED UNTIL I TERMINATE
Connection closed
Connection from: ('192.168.1.6', 49768)
Message from: ('192.168.1.6', 49768)
Received :
This is client 2
Sending:
THIS IS CLIENT 2
Connection closed

Command Prompt
C:\Users\berna\python>python client_tcp_simple.py
host -> localhost
port -> 4400
-> This is client 1
Received from server:
THIS IS CLIENT 1
-> Client 2 is blocked until I terminate
Received from server:
CLIENT 2 IS BLOCKED UNTIL I TERMINATE
-> bye
C:\Users\berna\python>

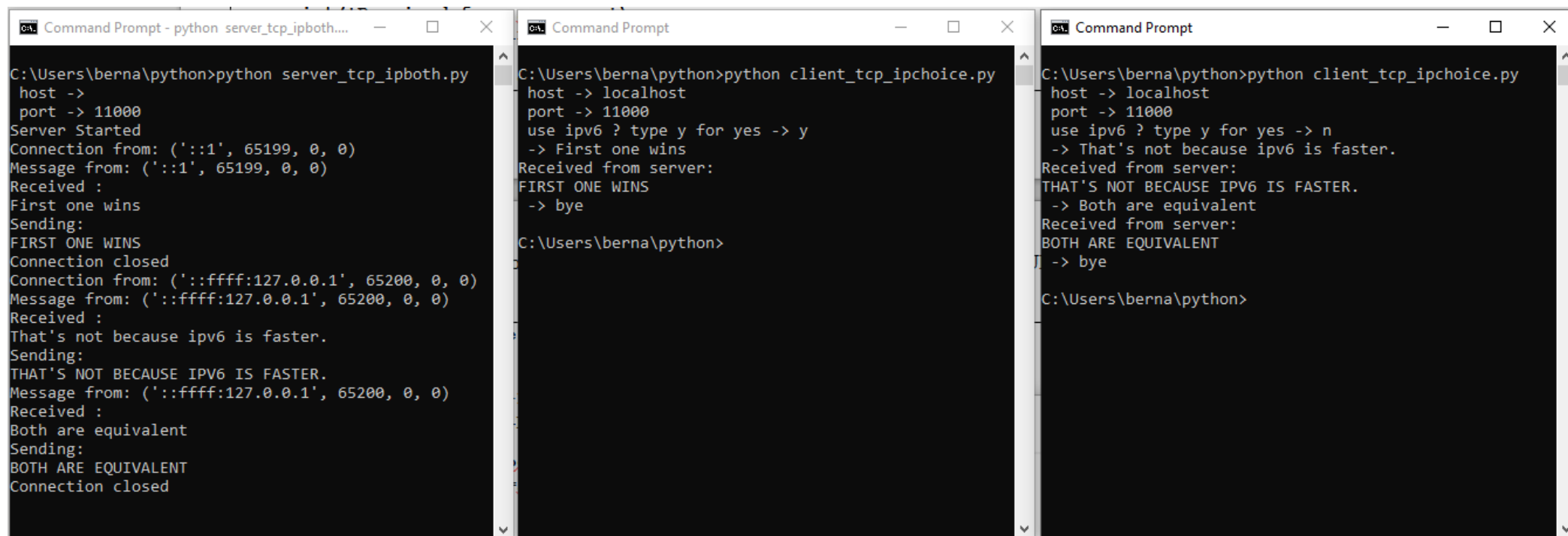
Command Prompt
C:\Users\berna\python>python client_tcp_simple.py
host -> 192.168.1.6
port -> 4400
-> This is client 2
Received from server:
THIS IS CLIENT 2
-> bye
C:\Users\berna\python>
```

Of course these codes can be adapted to allow the choice between IPv4 and IPv6 like in UDP

```
# place this code in file client_tcp_ipchoice.py :
import socket
host = input(" host -> ")
ipversion = input(" use ipv6 ? type y for yes -> ").lower()
ipv = socket.AF_INET
if ipversion=='y': ipv=socket.AF_INET6
server = (host,port)
client_socket = socket.socket(family=ipv,type=socket.SOCK_STREAM)
client_socket.connect(server) # connect to the server
message = input(" -> ") # take input
while message.lower().strip() != 'bye':
    client_socket.send(message.encode()) # send message
    data,addr = client_socket.recvfrom(1024)
    data=data.decode() # receive response
    print('Received from server: ')
    print(data) # show in terminal
    message = input(" -> ") # again take input
client_socket.close() # close the connection
```

And we'll go directly to the solution where the server accepts both IPv4 and IPv6 clients like in UDP.

```
# place this code in file server_tcp_ipboth.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket(family=socket.AF_INET6)
s.setsockopt(socket.IPPROTO_IPV6, socket.IPV6_V6ONLY, 0)
s.bind((host, port))
s.listen(1)
while True:
    conn, address = s.accept() # accept new connection
    print("Connection from: " + str(address))
    while True:
        data = conn.recv(1024).decode('utf-8')
        if not data: break
        print("Message from: " + str(address))
        print("Received : ")
        print(data)
        data = data.upper()
        print("Sending: ")
        print(data)
        conn.send(data.encode('utf-8'))
    conn.close()
    print("Connection closed")
s.close()
```



```

C:\Users\berna\python>python server_tcp_ipboth.py
host ->
port -> 11000
Server Started
Connection from: ('::1', 65199, 0, 0)
Message from: ('::1', 65199, 0, 0)
Received :
First one wins
Sending:
FIRST ONE WINS
Connection closed
Connection from: ('::ffff:127.0.0.1', 65200, 0, 0)
Message from: ('::ffff:127.0.0.1', 65200, 0, 0)
Received :
That's not because ipv6 is faster.
Sending:
THAT'S NOT BECAUSE IPV6 IS FASTER.
Message from: ('::ffff:127.0.0.1', 65200, 0, 0)
Received :
Both are equivalent
Sending:
BOTH ARE EQUIVALENT
Connection closed

C:\Users\berna\python>python client_tcp_ipchoice.py
host -> localhost
port -> 11000
use ipv6 ? type y for yes -> y
-> First one wins
Received from server:
FIRST ONE WINS
-> bye
C:\Users\berna\python>

C:\Users\berna\python>python client_tcp_ipchoice.py
host -> localhost
port -> 11000
use ipv6 ? type y for yes -> n
-> That's not because ipv6 is faster.
Received from server:
THAT'S NOT BECAUSE IPV6 IS FASTER.
-> Both are equivalent
Received from server:
BOTH ARE EQUIVALENT
-> bye
C:\Users\berna\python>

```

Once again it seems impossible to accept and respond to more than one client at the same time.
And once again the solution is quickly found in forums ...

Fortunately ... Python allows multithread programming, so the following one is a much better server ...
Praise be Python ... I don't know if it's so easy with another programming language :)

```

# place this code in file server_tcp_multi.py :
import socket
import threading

def on_new_client(conn,addr):
    while True:
        data = conn.recv(1024).decode('utf-8')
        if not data: break
        print("Message from: " + str(address))
        print("Received : ")
        print(data)
        data = data.upper()
        print("Sending: ")
        print(data)
        conn.send(data.encode('utf-8'))
    conn.close()
    print("Connection closed")

host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket(family=socket.AF_INET6)
s.setsockopt(socket.IPPROTO_IPV6,socket.IPV6_V6ONLY,0)
s.bind((host, port))
print("Server Started")
s.listen(1)
while True:
    conn, address = s.accept() # accept new connection
    print("Connection from: " + str(address))
    x = threading.Thread(target=on_new_client, args=(conn,address,))
    x.start()
s.close()

```

def is a procedure declaration

Each time the server receives a connection from a new client, it starts a new thread with the connection to that client ...

```
Command Prompt - python server_tcp_multi.py
C:\Users\berna\python>python server_tcp_multi.py
host ->
port -> 10500
Server Started
Connection from: ('::1', 65263, 0, 0)
Connection from: ('::ffff:192.168.1.6', 65266, 0, 0)
)
Message from: ('::1', 65263, 0, 0)
Received :
Now both client work together
Sending:
NOW BOTH CLIENT WORK TOGETHER
Message from: ('::ffff:192.168.1.6', 65266, 0, 0)
Received :
I love multithreading
Sending:
I LOVE MULTITHREADING
Message from: ('::1', 65263, 0, 0)
Received :
me too
Sending:
ME TOO
Connection closed with: ('::ffff:192.168.1.6', 65266, 0, 0)
Connection closed with: ('::1', 65263, 0, 0)

Command Prompt
C:\Users\berna\python>python client_tcp_ipchoice.py
host -> localhost
port -> 10500
use ipv6 ? type y for yes -> y
-> Now both client work together
Received from server:
NOW BOTH CLIENT WORK TOGETHER
-> me too
Received from server:
ME TOO
-> bye
C:\Users\berna\python>

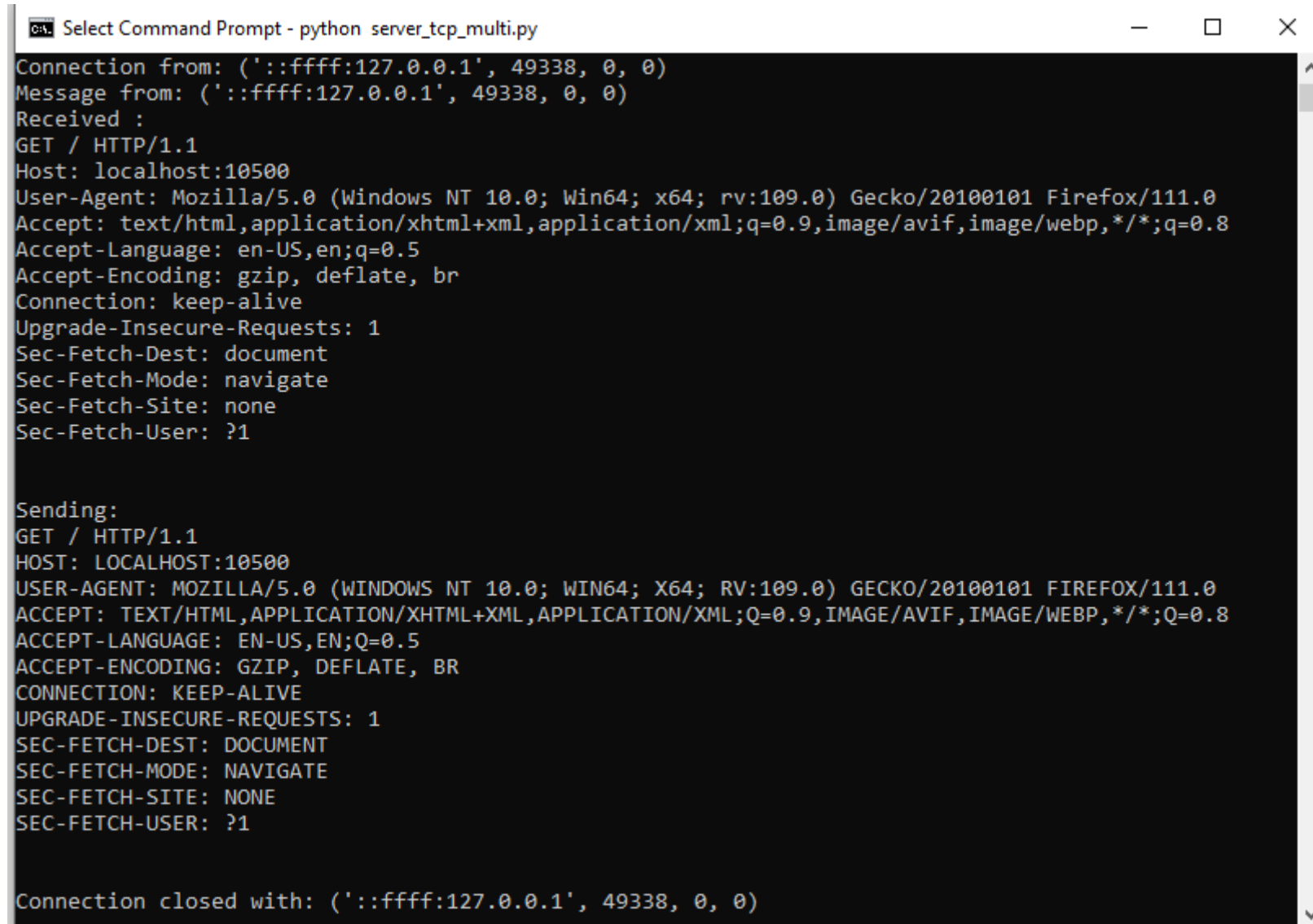
Command Prompt
C:\Users\berna\python>python client_tcp_ipchoice.py
host -> 192.168.1.6
port -> 10500
use ipv6 ? type y for yes -> n
-> I love multithreading
Received from server:
I LOVE MULTITHREADING
-> bye
C:\Users\berna\python>
```

Amazingly, by typing

localhost:10500

in the address bar of a browser ...

Of course our simple server received the request ... but as it is not an HTTP server, it didn't send an adequate answer (just echoing the browser request in upper case), so nothing happens on the browser window ...



```
Select Command Prompt - python server_tcp_multi.py
Connection from: (::ffff:127.0.0.1', 49338, 0, 0)
Message from: (::ffff:127.0.0.1', 49338, 0, 0)
Received :
GET / HTTP/1.1
Host: localhost:10500
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/111.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1

Sending:
GET / HTTP/1.1
HOST: LOCALHOST:10500
USER-AGENT: MOZILLA/5.0 (WINDOWS NT 10.0; WIN64; X64; RV:109.0) GECKO/20100101 FIREFOX/111.0
ACCEPT: TEXT/HTML,APPLICATION/XHTML+XML,APPLICATION/XML;Q=0.9,IMAGE/AVIF,IMAGE/WEBP,*/*;Q=0.8
ACCEPT-LANGUAGE: EN-US,EN;Q=0.5
ACCEPT-ENCODING: GZIP, DEFLATE, BR
CONNECTION: KEEP-ALIVE
UPGRADE-INSECURE-REQUESTS: 1
SEC-FETCH-DEST: DOCUMENT
SEC-FETCH-MODE: NAVIGATE
SEC-FETCH-SITE: NONE
SEC-FETCH-USER: ?1

Connection closed with: (::ffff:127.0.0.1', 49338, 0, 0)
```

And now the final test ... run the server on one computer and the client on an other one.

Be careful of a few things ...

- Your computer firewall will probably block the connection, so you can :

- Either temporarily disable your firewall (not recommended if you are connected on internet)

- Or set the connection to PRIVATE (it's in the connection settings), that means that your firewall will assume that any connection within your LAN is secure

- Most routers, including ISP boxes are capable of IPv6 routing but not by default within hosts on your LAN. Moreover they don't have a DNS translation in IPv6.

Only IPv4 translation. You can check that by typing

nslookup <host name of your computer>

and hostname of your computer can be get with

whoami

So if you want to use IPv6, you have to type the complete IPv6 address it received from the LAN DHCP server.

A good solution is always to ping the other computer before trying ... if ping works then python sockets will work too...

Here are a few things I tested ...

First this is ipconfig of both machines :

ipconfig from client

ipconfig from server

```
Ethernet adapter Ethernet:

Connection-specific DNS Suffix . : dlink
IPv6 Address. . . . . : fd4d:3aac:67a0:0:c5b0:442e:e4c0:23c2
Temporary IPv6 Address. . . . . : fd4d:3aac:67a0:0:1ff:fbf9:e5a6:3872
Link-local IPv6 Address . . . . . : fe80::2804:30a6:34b9:6459%13
IPv4 Address. . . . . : 192.168.0.100
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.0.1

Wireless LAN adapter WiFi:

Connection-specific DNS Suffix . : lan
IPv6 Address. . . . . : 2a02:a03f:c0d8:2800:3314:2513:a5fa:6f86
IPv6 Address. . . . . : fd9c:2334:d3a5:0:55c8:4948:9cea:7944
Temporary IPv6 Address. . . . . : 2a02:a03f:c0d8:2800:dc1a:4a8e:cb61:426d
Temporary IPv6 Address. . . . . : fd9c:2334:d3a5:0:dc1a:4a8e:cb61:426d
Link-local IPv6 Address . . . . . : fe80::a2ad:e0ad:c3c3:b48b%19
IPv4 Address. . . . . : 192.168.1.6
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : fe80::a6b1:e9ff:febd:5b12%19
                          192.168.1.1
```

```
Carte Ethernet Ethernet :

Suffixe DNS propre à la connexion. . . : dlink
Adresse IPv6. . . . . : fd4d:3aac:67a0:0:cc8a:105e:ae6d:afe0
Adresse IPv6 temporaire . . . . . : fd4d:3aac:67a0:0:6d2c:f202:c723:a7b4
Adresse IPv6 de liaison locale. . . . : fe80::cc8a:105e:ae6d:afe0%15
Adresse IPv4. . . . . : 192.168.0.101
Masque de sous-réseau. . . . . : 255.255.255.0
Passerelle par défaut. . . . . : 192.168.0.1

Carte réseau sans fil Wi-Fi :

Suffixe DNS propre à la connexion. . . : lan
Adresse IPv6. . . . . : 2a02:a03f:c0d8:2800:fc99:ed4:7644:2522
Adresse IPv6. . . . . : fd9c:2334:d3a5:0:fc99:ed4:7644:2522
Adresse IPv6 temporaire . . . . . : 2a02:a03f:c0d8:2800:48a9:6174:f57a:e610
Adresse IPv6 temporaire . . . . . : fd9c:2334:d3a5:0:48a9:6174:f57a:e610
Adresse IPv6 de liaison locale. . . . : fe80::fc99:ed4:7644:2522%4
Adresse IPv4. . . . . : 192.168.1.13
Masque de sous-réseau. . . . . : 255.255.255.0
Passerelle par défaut. . . . . : fe80::a6b1:e9ff:febd:5b12%4
                          192.168.1.1
```

Both machines are connected to 2 different routers.

- One through Ethernet cable, it's a Dlink DIR-910 router with no WAN connection to internet during test. This is an old router (bios dating 2013)
- One through WiFi, its a bbox3 from ISP

First note that both interfaces provide different IP addresses.

Both routers provides DHCP in v4 and in v6.

Only bbox provides additional information like DNS, but we will see that bbox is unable to route LAN in v6 while old Dlink can.

Test sessions on client

Transcript on server

```
C:\Users\berna\python>python client_tcp_ipchoice.py
host -> LAPTOP-K9MVJBE2
port -> 3000
use ipv6 ? type y for yes -> n
-> Test
Received from server:
TEST
-> bye

C:\Users\berna\python>python client_tcp_ipchoice.py
host -> 192.168.0.101
port -> 3000
use ipv6 ? type y for yes -> n
-> Test2
Received from server:
TEST2
-> bye

C:\Users\berna\python>python client_tcp_ipchoice.py
host -> fd4d:3aac:67a0:0:cc8a:105e:ae6d:afe0
port -> 3000
use ipv6 ? type y for yes -> y
-> test3
Received from server:
TEST3
-> bye

C:\Users\berna\python>python client_tcp_ipchoice.py
host -> 2a02:a03f:c0d8:2800:fc99:ed4:7644:2522
port -> 3000
use ipv6 ? type y for yes -> y
Traceback (most recent call last):
  File "C:\Users\berna\python\client_tcp_ipchoice.py", line 9, in <module>
    client_socket.connect(server) # connect to the server
TimeoutError: [WinError 10060] A connection attempt failed because the co
nnected party did not properly respond after a period of time, or establi
shed connection failed because connected host has failed to respond

C:\Users\berna\python>_
```

```
D:\Python>python server_tcp_multi.py
host ->
port -> 3000
Server Started
Connection from: ('::ffff:192.168.1.6', 61234, 0, 0)
Message from: ('::ffff:192.168.1.6', 61234, 0, 0)
Received :
Test
Sending:
TEST
Connection closed with: ('::ffff:192.168.1.6', 61234, 0, 0)
Connection from: ('::ffff:192.168.0.100', 61315, 0, 0)
Message from: ('::ffff:192.168.0.100', 61315, 0, 0)
Received :
Test2
Sending:
TEST2
Connection closed with: ('::ffff:192.168.0.100', 61315, 0, 0)
Connection from: ('fd4d:3aac:67a0:0:1ff:fbf9:e5a6:3872', 61373, 0, 0)
Message from: ('fd4d:3aac:67a0:0:1ff:fbf9:e5a6:3872', 61373, 0, 0)
Received :
test3
Sending:
TEST3
Connection closed with: ('fd4d:3aac:67a0:0:1ff:fbf9:e5a6:3872', 61373, 0, 0)
```


1. Test in IPv4 while providing server hostname. We see that bbox makes DNS translation, on server received address is 192.168.1.13
2. Test in IPv4 with hard IP address on Dlink server.
3. Test in IPv6 with hard IPv6 address on Dlink server
4. Test in IPv6 with hard IPv6 address on bbox server. This one fails and crashes the client.

Of course all of you are encouraged to test with your own hardware ...

And ...

=> **Mandatory homework : report on this lab**

Test a ~~client~~/server system in Python

You can replace the ~~transformation~~ to uppercase by the server with something else. A good exercise should be for example to send a small text file or simulate a small 2-players game.

You can go further that was presented here

You can compare the possibilities with your own hardware and/or OS, ~~as everything here is dependent on that~~

There is no obligation of result. If you fail, then just explain what you think is ~~the problem.~~

This should be real report (not only source files)

Deadline : before course session 7 (2 weeks)

2024 :

Write a network application : for example a multiplayer game

- You can use the client/server paradigm (which is probably easier) or peer-to-peer
- You can use TCP or UDP sockets (which is probably easier) or any other access to lower layer protocols
- Programming language should be python
- Your application must work at least on distributed machines within the same LAN
- Deliverables are a full pdf report plus the source code
- Work can be done individually or by groups of 2 or 3 maximum (the more in the group the higher are expectations)
- Deadline is Sunday before the break at the end of semester (in 2024 : June 2)