

Network Programming with Python

Sébastien Tixeuil
sebastien.Tixeuil@lip6.fr

Python 101

Python Standard Types

- Numbers: 1, 122L, 2.0, 1e+2j
- Strings: 'Python', 'PROGRES'
- Lists: [123, 'ABC']
- Tuples: (123, 'ABC')
- Dictionaries: {'num': 123, 'str': 'ABC'}

Python Subscripting

```
str = "PROGRES"
print(str[2:4])
print(str[2:])
print(str[-1:])

l = [1,2,3,4,5]
print(l[2:4])
print(l[2:])
print(l[-1:])
```

Python Subscripting

```
str = "PROGRES"
print(str[2:4]) → OG
print(str[2:]) → OGRES
print(str[-1:]) → S

l = [1,2,3,4,5]
print(l[2:4]) → [3, 4]
print(l[2:]) → [3, 4, 5]
print(l[-1:]) → [5]
```

Python Operators

- Arithmetic: +, -, *, /, %, **, //
- Comparison: ==, !=, <, >, <=, >=
- Assignment: =, +=, -=, *=, /=, %=, **=, //=
- Logical: and, or, not
- Bitwise: &, |, ~, <<, >>
- Membership: in, not in
- Identity: is, is not

Python Conditions

```
var1 = 100
if var1:
    print ("1 - Got a true expression value")
    print (var1)
else:
    print ("1 - Got a false expression value")
    print (var1)
var2 = 0
if var2:
    print ("2 - Got a true expression value")
    print (var2)
else:
    print ("2 - Got a false expression value")
    print (var2)
print ("Good bye!")
```

Python Conditions

```
var1 = 100
if var1:
    print ("1 - Got a true expression value")
    print (var1) → 1 - Got a true expression value
else:
    print ("1 - Got a false expression value") → 100
    print (var1)
var2 = 0
if var2:
    print ("2 - Got a true expression value")
    print (var2)
else:
    print ("2 - Got a false expression value") → 2 - Got a false expression value
    print (var2) → 0
    print ("Good bye!") → Good bye!
print ("Good bye!")
```

Python While Loop

```
count = 0

while (count < 9):

    print ('The count is:', count)

    count = count + 1

print ("Good bye!")
```

Python While Loop

```
count = 0

while (count < 9):

    print ('The count is:', count)

    count = count + 1

print ("Good bye!")
```

The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!

Python For Loop

```
for letter in 'Python':

    print ('Current Letter :', letter)

fruits = ['banana', 'apple', 'mango']

for fruit in fruits:

    print ('Current fruit :', fruit)

fruits = ['banana', 'apple', 'mango']

for index in range(len(fruits)):

    print ('Current fruit :', fruits[index])
```

Python For Loop

```
for letter in 'Python':

    print ('Current Letter :', letter)

fruits = ['banana', 'apple', 'mango']

for fruit in fruits:

    print ('Current fruit :', fruit)

fruits = ['banana', 'apple', 'mango']

for index in range(len(fruits)):

    print ('Current fruit :', fruits[index])
```

Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Current fruit : banana
Current fruit : apple
Current fruit : mango

Python Exceptions

try:

// dangerous code

except ExceptionName:

// code to handle the problem related

// to the ExceptionName event

Python Exceptions

```
from sys import *

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Python Exceptions

try:

raise Exception('prog','res')

except Exception as inst:

print(type(inst))

print(inst.args)

print(inst)

x, y = inst.args

print('x =', x)

print('y =', y)

Python Exceptions

try:

raise Exception('prog','res')

except Exception as inst:

print(type(inst))

print(inst.args)

print(inst)

x, y = inst.args

print('x =', x)

print('y =', y)

<class 'Exception'>

('prog', 'res')

('prog', 'res')

x = prog

y = res

Python Functions

```
def sum(arg1,arg2):  
    res = arg1+arg2  
  
    return res
```

```
print(sum(2,2))
```

Python Functions

```
def sum(arg1,arg2):  
    res = arg1+arg2  
  
    return res
```

```
print(sum(2,2)) —————→ 4
```

Python Functions

```
res=0
```

```
def sum(arg1,arg2):  
    res = arg1+arg2  
  
    return res
```

```
print(sum(2,2))
```

```
print(res)
```

Python Functions

```
res=0
```

```
def sum(arg1,arg2):  
    res = arg1+arg2  
  
    return res
```

```
print(sum(2,2)) —————→ 4
```

```
print(res) —————→ 0
```

Python Functions

```
def reassign(list):  
    list = [0, 1]
```

```
def append(list):  
    list.append(1)
```

```
list = [0]  
print(list)  
reassign(list)  
print(list)  
append(list)  
print(list)
```

Python Functions

```
def reassign(list):  
    list = [0, 1]
```

```
def append(list):  
    list.append(1)
```

```
list = [0]  
print(list)  
reassign(list)  
print(list)  
append(list)  
print(list)
```

The diagram illustrates the state of the `list` variable and the list object it points to. It shows three stages of execution: 1. `list = [0]` and `print(list)` output `[0]`. 2. `reassign(list)` is called, but `list` still points to `[0]`. 3. `append(list)` is called, and `list` now points to a new list object `[0, 1]`, which is printed.

Python Modules

myfuncs.py

```
def add(arg1,arg2):  
    return arg1+arg2
```

usemyfuncs.py

```
import myfuncs  
print(myfuncs.add(2,2))
```

Python Modules

myfuncs.py

```
def add(arg1,arg2):  
    return arg1+arg2
```

usemyfuncs.py

```
from myfuncs import *  
print(add(2,2))
```

Client-Server

Addressing

Client

Server

A.B.C.D

W.X.Y.Z

Addressing

Client

Server

Process 1

Process 1

Process 2

Process 2

A.B.C.D

W.X.Y.Z

Client-Server

Client

Server

Process 1

Service 1
port: 1234

Process 2

Service 2
port: 4567

A.B.C.D

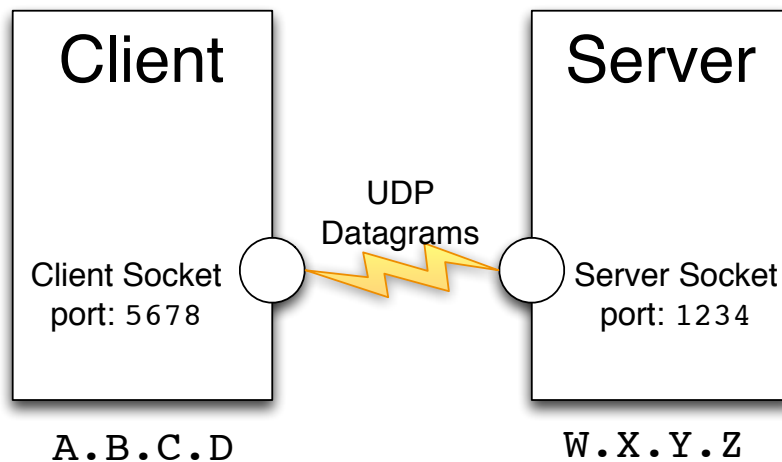
W.X.Y.Z

Client-Server

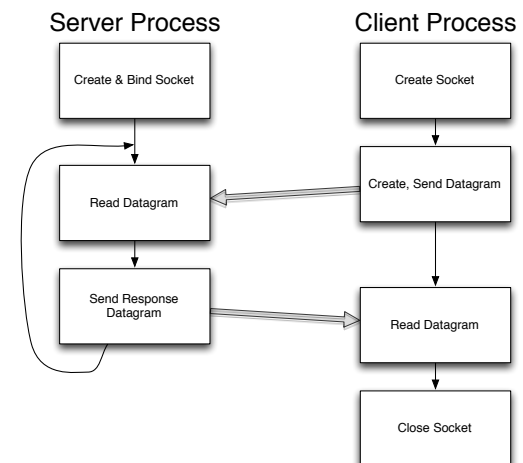
- Two processes, **client** and **server**, are executed on (possibly) different machines
- The server **awaits** connections and provides a service
- The client **connects** to the server to request a service
- A particular process can be both client and server (possibly) for different services

UDP Sockets

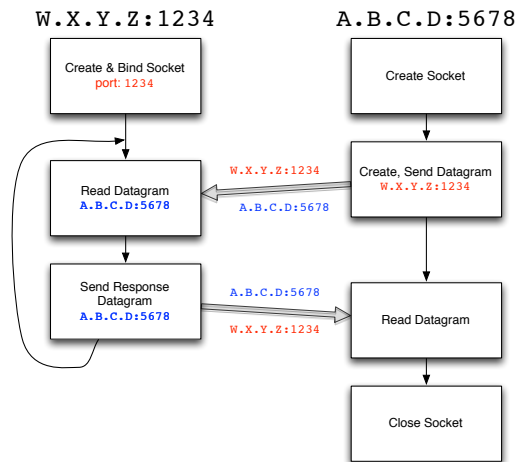
UDP Sockets



UDP Client-Server



UDP Client-Server



UDP Python Client

```

from socket import *

serverName = 'A.B.C.D'

serverPort = 1234

clientSocket = socket(AF_INET, SOCK_DGRAM)

message = input('lowercase sentence:')

clientSocket.sendto(message, (serverName, serverPort))

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

print(modifiedMessage)

clientSocket.close()
  
```

UDP Python Server

```

from socket import *

serverPort = 1234

serverSocket = socket(AF_INET, SOCK_DGRAM)

serverSocket.bind(('', serverPort))

print('server ready')

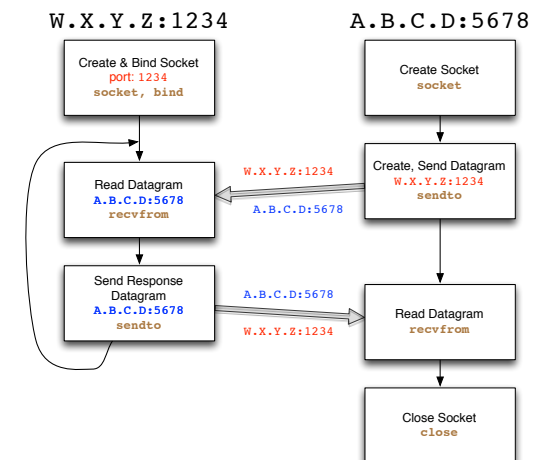
while True:

    message, clientAddress = serverSocket.recvfrom(2048)

    modifiedMessage = message.upper()

    serverSocket.sendto(modifiedMessage, clientAddress)
  
```

UDP Client-Server



Strings and Bytes

Strings vs. Bytes

- **Strings** are meant for general Unicode support in Python 3
- **Bytes** are what is sent/received through the network
- **Encoding** of Strings into Bytes before sending
`toSend = str.encode('utf-8')`
- **Decoding** Bytes into Strings when receiving
`str = received.decode('utf-8')`

UDP Python Client

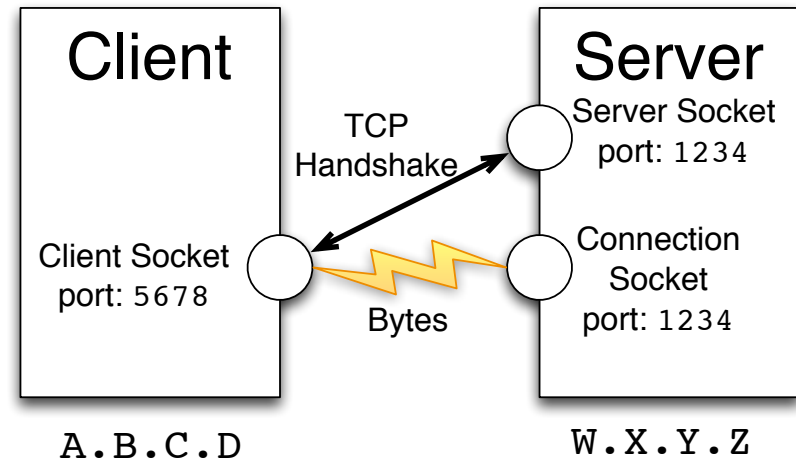
```
from socket import *
serverName = 'A.B.C.D'
serverPort = 1234
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input('lowercase sentence:').encode('utf-8')
clientSocket.sendto(message, (serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print(modifiedMessage.decode('utf-8'))
clientSocket.close()
```

UDP Python Server

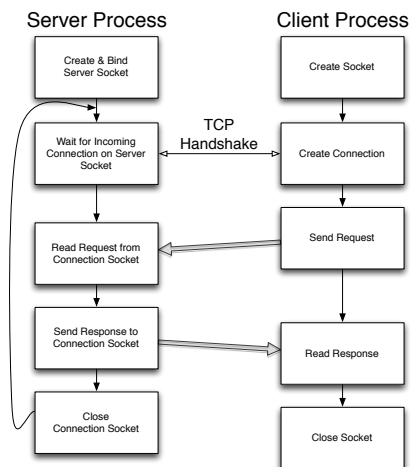
```
from socket import *
serverPort = 1234
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print('server ready')
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode('utf-8').upper()
    serverSocket.sendto(modifiedMessage.encode('utf-8'),
                        clientAddress)
```

TCP Sockets

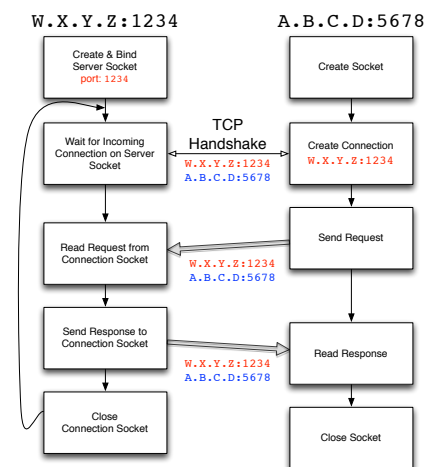
TCP Sockets



TCP Client-Server



TCP Client-Server



TCP Python Client

```
from socket import *

serverName = 'A.B.C.D'

serverPort = 1234

clientSocket = socket(AF_INET, SOCK_STREAM)

clientSocket.connect((serverName, serverPort))

message = input('lowercase sentence:')

clientSocket.send(message)

modifiedMessage = clientSocket.recv(2048)

print(modifiedMessage)

clientSocket.close()
```

TCP Python Server

```
from socket import *

serverPort = 1234

serverSocket = socket(AF_INET, SOCK_STREAM)

serverSocket.bind(('', serverPort))

serverSocket.listen(1)

print('server ready')

while True:

    connectionSocket, address = serverSocket.accept()

    message = connectionSocket.recv(2048)

    modifiedMessage = message.upper()

    connectionSocket.send(modifiedMessage)

    connectionSocket.close()
```

TCP Python Client

```
from socket import *

serverName = 'A.B.C.D'

serverPort = 1234

clientSocket = socket(AF_INET, SOCK_STREAM)

clientSocket.connect((serverName, serverPort))

message = input('lowercase sentence:').encode('utf-8')

clientSocket.send(message)

modifiedMessage = clientSocket.recv(2048).decode('utf-8')

print(modifiedMessage)

clientSocket.close()
```

TCP Python Server

```
from socket import *

serverPort = 1234

serverSocket = socket(AF_INET, SOCK_STREAM)

serverSocket.bind(('', serverPort))

serverSocket.listen(1)

print('server ready')

while True:

    connectionSocket, address = serverSocket.accept()

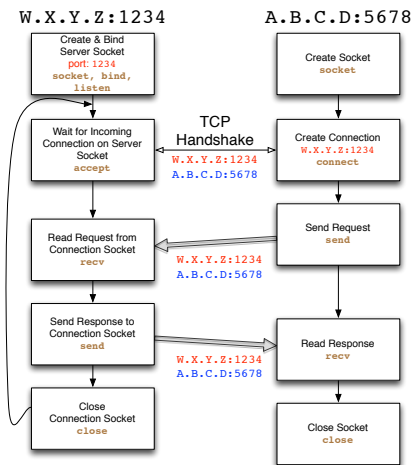
    message = connectionSocket.recv(2048).decode('utf-8')

    modifiedMessage = message.upper().encode('utf-8')

    connectionSocket.send(modifiedMessage)

    connectionSocket.close()
```

TCP Client Server



IPv4 and IPv6

IPv4 Names

```

from socket import *

print(gethostname())

print(getfqdn())

print(gethostbyname('lip6.fr'))

print(gethostbyaddr('132.227.104.15'))

print(gethostbyname(getfqdn()))
  
```

IPv4 Names

```

from socket import *

print(gethostname())
print(getfqdn())

print(gethostbyname('lip6.fr'))

print(gethostbyaddr('132.227.104.15'))

print(gethostbyname(getfqdn()))
  
```

The diagram shows the output of the IPv4 names program. Arrows point from the print statements to their respective outputs:

- `print(gethostname())` points to `postetixeuil4.rsr.lip6.fr`
- `print(getfqdn())` points to `postetixeuil4.rsr.lip6.fr`
- `print(gethostbyname('lip6.fr'))` points to `132.227.104.15`
- `print(gethostbyaddr('132.227.104.15'))` points to `('ww.lip6.fr', ['15.104.227.132.in-addr.arpa'], ['132.227.104.15'])`
- `print(gethostbyname(getfqdn()))` points to `132.227.84.244`

IPv4-IPv6 Names

```
infolist = getaddrinfo('lip6.fr', 'www')

print(infolist)

info = infolist[1]

print(info)

s = socket(*info[0:3])

s.connect(info[4])
```

IPv4-IPv6 Names

```
infolist = getaddrinfo('lip6.fr', 'www')

print(infolist)

info = infolist[1]

print(info)

s = socket(*info[0:3])

s.connect(info[4])
```

Diagram illustrating the output of `getaddrinfo` for the domain `lip6.fr`. The output is a list of tuples, each representing an address family, socket kind, and address. The first tuple is `(<AddressFamily.AF_INET: 2>, <SocketKind.SOCK_DGRAM: 2>, 17, '', ('132.227.104.15', 80))`. The second tuple is `(<AddressFamily.AF_INET: 2>, <SocketKind.SOCK_STREAM: 1>, 6, '', ('132.227.104.15', 80))`. Arrows point from the `print(infolist)` and `print(info)` lines to these respective tuples.

Numbers and Byte Order

```
from struct import *

print(hex(1234))

print(pack('<i', 1234))
print(pack('>i', 1234))
print(pack('!i', 1234))

print(unpack('>i', b'\x00\x00\x04\xd2'))

print(unpack('!i', b'\x00\x00\x04\xd2'))
```

Byte Order over the Network

Byte Order over the Network

```
from struct import *

print(hex(1234)) → 0x4d2

print(pack('<i', 1234)) → b'\xd2\x04\x00\x00'
print(pack('>i', 1234)) → b'\x00\x00\x04\xd2'
print(pack('!i', 1234)) → b'\x00\x00\x04\xd2'

print(unpack('>i', b'\x00\x00\x04\xd2')) → (1234,)
print(unpack('!i', b'\x00\x00\x04\xd2')) → (1234,)
```

Network Exceptions

Network Exceptions

- **OSError**: almost every failure that can happen during a network connection
- **socket.gaierror**: address-related error
- **socket.timeout**: timeout expired

Network Exceptions

```
from socket import *

try:

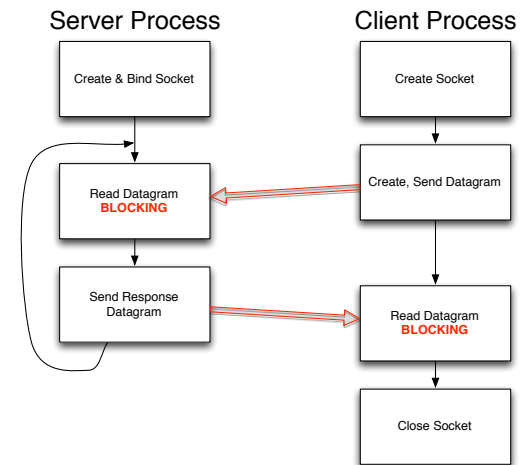
    infolist = getaddrinfo('nonexistent.com', 'www')

except gaierror:

    print("This host does not seem to exist")
```

UDP Packet Drops

UDP Packet Drops



UDP Packet Drops

```
...
delay = 0.1 # sec
while True:
    clientSocket.sendto(message, (serverName, serverPort))
    clientSocket.settimeout(delay)
    try:
        modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
    except socket.timeout:
        delay *= 2
    else:
        break
print(modifiedMessage.decode('utf-8'))
clientSocket.close()
```

UDP Packet Drops

```
...
delay = 0.1 # sec
while True:
    clientSocket.sendto(message, (serverName, serverPort))
    clientSocket.settimeout(delay)
    try:
        modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
    except socket.timeout:
        delay *= 2
        if delay > 2.0:
            raise RuntimeError('server seems down')
    else:
        break
print(modifiedMessage.decode('utf-8'))
```


UDP Broadcast

UDP Broadcast Client

```
from socket import *

broadcastAddr = 'W.X.Y.255' # assuming 255.255.255.0 netmask

serverPort = 1234

clientSocket = socket(AF_INET, SOCK_DGRAM)

clientSocket.setsockopt(SOL_SOCKET, SO_BROADCAST, 1)

message = input('lowercase sentence:').encode('utf-8')

clientSocket.sendto(message, (serverName, serverPort))

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

print(modifiedMessage.decode('utf-8'))

clientSocket.close()
```

UDP Broadcast Server

```
from socket import *

serverPort = 1234

serverSocket = socket(AF_INET, SOCK_DGRAM)

serverSocket.bind(('', serverPort))

print('server ready')

while True:

    message, clientAddress = serverSocket.recvfrom(2048)

    modifiedMessage = message.decode('utf-8').upper()

    serverSocket.sendto(modifiedMessage.encode('utf-8'),
        clientAddress)
```

TCP Sending

TCP send

- If TCP/IP stack has enough room, **send** returns immediately, and the **complete** message will be handled
- If TCP/IP stack is full, **send** is **blocking**
- If TCP/IP stack is **almost** full, **send** returns immediately, but only a **part** of the message will be handled

TCP send Loop

```
...  
message=input('lowercase sentence:').encode('utf-8')  
bytes_sent = 0  
while bytes_sent < len(message):  
    message_remaining = message[bytes_sent:]  
    bytes_sent +=  
        clientSocket.send(message_remaining)  
...
```

TCP sendall

```
...  
message = input('lowercase  
sentence:').encode('utf-8')  
clientSocket.sendall(message)  
...
```

TCP Receiving

TCP `recv`

- If TCP/IP stack is full enough, `recv` returns immediately, and the **specified size** message will be delivered
- If TCP/IP stack is empty, `recv` is **blocking**
- If TCP/IP stack is not empty, `recv` returns immediately, but only a **fraction** of the specified size will be delivered

TCP `recvall` ?

- Deciding when **all** data is received is **application specific**
 - Fixed size messages
 - Message size is announced before data is sent
 - Special delimiters announce end of data

Framing and Quoting

TCP `recvall`

```
def recvall(sock, length):  
    blocks = []  
    while length:  
        block = sock.recv(length)  
        if not block:  
            raise EOFError('socket closed with %d bytes left'  
                            ' in this block'.format(length))  
        length -= len(block)  
        blocks.append(block)  
    return b''.join(blocks)
```

Sending a Block

```
from struct import *
from socket import *

header_struct = Struct('!I')

def put_block(sock, message):

    block_length = len(message)

    sock.sendall(header_struct.pack(
        block_length))

    sock.sendall(message)
```

Receiving a Block

```
from struct import *
from socket import *

header_struct = Struct('!I')

def get_block(sock):

    data = recvall(sock, header_struct.size)

    (block_length,) = header_struct.unpack(data)

    return recvall(sock, block_length)
```

Files and Directories

Pathlib

- Possible issues when dealing with files :
 - OS conventions: \ vs /
 - Absolute vs. relative
 - Search for specific files
 - Concatenation

Pathlib

- Useful methods: `exists`, `is_file`, `is_dir`, ...
- Simple Construction:
 - `Path('tempdir')` / `Path('tempfile')`

Pathlib

```
from pathlib import Path

d = Path('.')

file = Path('test.txt')

if d.is_file():
    print(f"{d} is a file.")

elif d.is_dir():
    print(f"{d} is a dir.")

if file.is_file():
    print(f"{file} is a file.")

elif file.is_dir():
    print(f"{file} is a dir.")
```

Pathlib

```
from pathlib import Path

d = Path('.')

file = Path('test.txt')

if d.is_file():
    print(f"{d} is a file.")

elif d.is_dir():
    print(f"{d} is a dir.")

if file.is_file():
    print(f"{file} is a file.")

elif file.is_dir():
    print(f"{file} is a dir.")
```

. is a dir.

test.txt is a file.

Pathlib

```
from pathlib import Path

d = Path('.')

file = Path('test.txt')

if d.is_file():
    print(f"{d} is a file.")

elif d.is_dir():
    print(f"{d} is a dir.")

if file.is_file():
    print(f"{file} is a file.")

elif file.is_dir():
    print(f"{file} is a dir.")
```

. is a dir.

test.txt is a file.

Pathlib

```
d = Path('.')

for file in d.rglob('*'):

    if file.is_file():

        print(f"{file} is a file.")

    elif file.is_dir():

        print(f"{file} is a dir.")
```

Pathlib

```
d = Path('.')

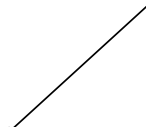
for file in d.rglob('*'):

    if file.is_file():

        print(f"{file} is a file.")

    elif file.is_dir():

        print(f"{file} is a dir.")
```



.DS_Store is a file.
d1 is a dir.
d2 is a dir.
test.txt is a file.
.ipynb_checkpoints is a dir.
pathlib.ipynb is a file.
d2/.DS_Store is a file.
d2/d21 is a dir.
.ipynb_checkpoints/pathlib-checkpoint.ipynb is a file.

Context Managers

```
def get_txt():

    f = open('test.txt', 'rb')

    txt = f.read().decode('utf8')

    f.close()

    return txt

get_txt()
```

Context Managers

```
def get_txt():

    with open('test.txt', 'rb') as f:

        return f.read().decode('utf8')

get_txt()
```

Compression

Python Data Compression

```
from zlib import *
```

```
str = b'A very long test string to evaluate compression and how much it improves  
bandwidth usage'
```

```
print(len(str))
```

```
data = compress(str)
```

```
print(data)
```

```
print(len(data))
```

```
d = decompressobj()
```

```
newdata = d.decompress(data)
```

```
print(newdata)
```

Python Data Compression

```
from zlib import *
```

```
str = b'A very long test string to evaluate compression and how much it improves  
bandwidth usage'
```

```
print(len(str))
```

88

```
data = compress(str)
```

```
print(data)
```

```
print(len(data))
```

```
b'x\x9c\x15\xc9\xd1\t\xc0  
\x0c\x05\xc0U\xde*\x1d%\xd5\xa0\x01%  
\x89J\xb7\xaf\xfd;  
\xb8\x0b\x8b\xedE\xd3Q\x10\xec\x01\x0f\x93\xdf\  
n^\xd4&\x05#i\x7f\x8c\xde\x07hdT\xdd\xe83UH@N\  
xe9b\xc7}fK\x8e\x8a\xe9T\xf8\x03\xad\\!\x05'
```

78

```
d = decompressobj()
```

```
newdata = d.decompress(data)
```

```
print(newdata)
```

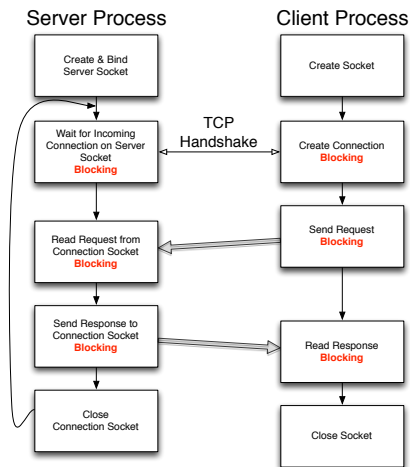
```
b'A very long test string to evaluate  
compression and how much it improves bandwidth  
usage'
```

Server-side Programming

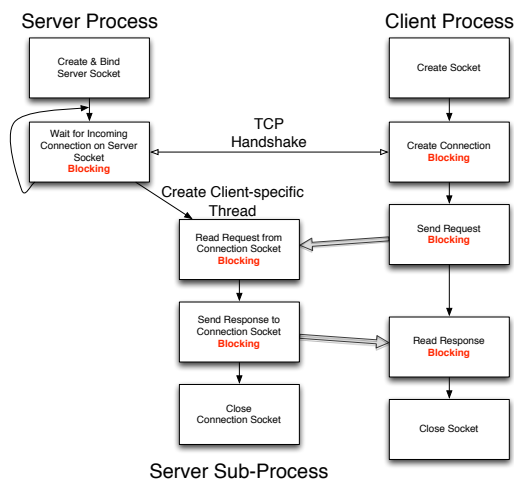
Sequential TCP Server

```
...  
  
def handle_client(clientSocket):  
    while True:  
        received = clientSocket.recv(4096)  
        if not received:  
            clientSocket.close()  
        else:  
            to_send = received.decode('utf-8').upper().encode('utf-8')  
            clientSocket.sendall(to_send)  
  
while True:  
    connectionSocket, address = serverSocket.accept()  
    handle_client(connectionSocket)
```

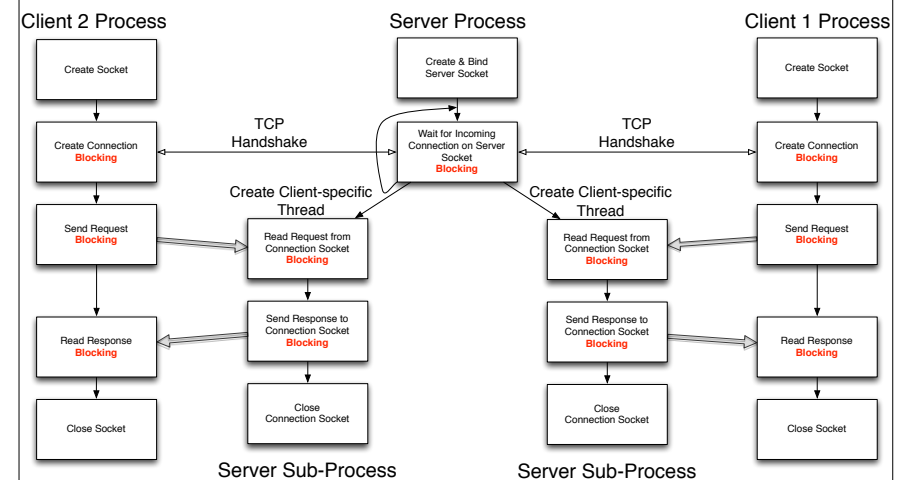
Sequential TCP Server



Multi-thread TCP Server



Multi-thread TCP Server



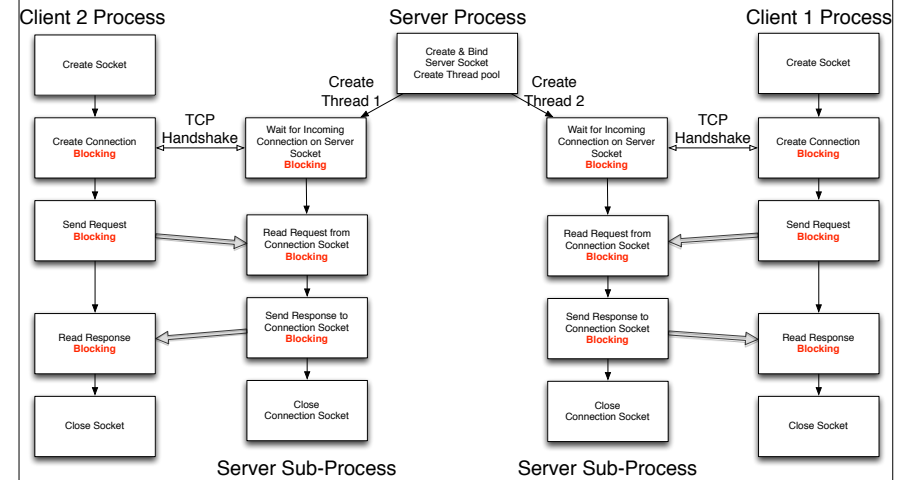
Multi-Thread TCP Server

```

from threading import *
...
def handle_client(clientSocket):
    while True:
        received = clientSocket.recv(4096)
        if not received:
            clientSocket.close()
        else:
            to_send = received.decode('utf-8').upper().encode('utf-8')
            clientSocket.sendall(to_send)
    while True:
        connectionSocket, address = serverSocket.accept()
        Thread(target=handle_client,args=(connectionSocket,)).start()

```

Thread Pool TCP Server



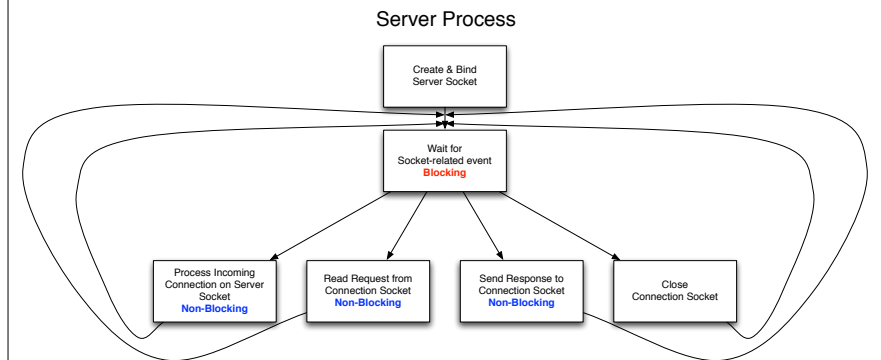
Thread Pool TCP Server

```

...
def handle_client(listeningSocket):
    while True:
        clientSocket, address = listeningSocket.accept()
        while True:
            received = clientSocket.recv(4096)
            if not received:
                clientSocket.close()
            else:
                to_send = received.decode('utf-8').upper().encode('utf-8')
                clientSocket.sendall(to_send)
    for i in range(4):
        Thread(target=handle_client,args=(serverSocket,)).start()

```

Multiplexing TCP Server



Multiplexing TCP Server

```
from select import *

...

my_poll = poll()
my_poll.register(serverSocket, POLLIN)

sockets = {serverSocket.fileno(): serverSocket}

# retrieve socket object from fileno
received = dict()

# bytes received from fileno, that are not yet processed
to_send = dict()

# bytes to be sent from fileno, that have been processed
```

Multiplexing TCP Server

```
while True:

    for fd, event in my_poll.poll():

        if event & (POLLHUP|POLLERR|POLLNVAL):

            received.pop(fd)

            to_send.pop(fd)

            my_poll.unregister(fd)

            del sockets[fd]

            sockets.pop(fd)
```

Multiplexing TCP Server

```
elif sockets[fd] is serverSocket:

    connectionSocket, address =
serverSocket.accept()

sockets[connectionSocket.fileno()] =
connectionSocket

my_poll.register(connectionSocket,
POLLIN)
```

Multiplexing TCP Server

```
else:

    if event & POLLIN:

        data = sockets[fd].recv(4096)

        if not data:

            sockets[fd].close()

            continue

        if fd in received:

            received[fd] += data

        else:

            received[fd] = data

        my_poll.modify(fd, POLLIN|POLLOUT)
```

Multiplexing TCP Server

```
if event & POLLOUT:

    data = received.pop(fd).decode('utf-8')
    data = data.upper().encode('utf-8')

    if fd in to_send:

        data = to_send.pop(fd) + data
    n = sockets[fd].send(data)

    if n < len(data):

        to_send[fd] = data[n:]
    else:

        my_poll.modify(fd,POLLIN)
```

Multiplexing TCP Server

```
from select import *

...
while True:

    for fd, event in my_poll.poll():

        if event & (POLLHUP|POLLERR|POLLNVAL):

            ...

        elif sockets[fd] is serverSocket:

            ...

        else:

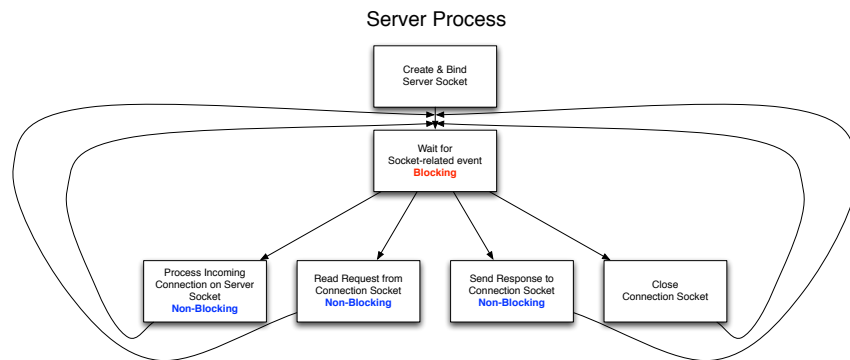
            if event & POLLIN:

                ...

            if event & POLLOUT:

                ...
```

Multiplexing TCP Server



XML-RPC

Remote Procedure Call

- The main objective is to make client-server calls (almost) transparent to the programmer
- The server **defines a set of functions** and makes them available through the network
- The client **calls the function** (almost) as if they were local
- No need to define a protocol, a data format, etc.

Python XML-RPC Server

```
from operator import *

from math import *

from xmlrpc.server import *

from functools import *

def addtogether(*things):

    return reduce(add,things)

def quadratic(a,b,c):

    b24ac = sqrt(b*b - 4.0*a*c)

    return list((set([(-b-b24ac)/2.0*a,(-b+b24ac)/2.0*a])))
```

Python XML-RPC Server

```
...

server = SimpleXMLRPCServer(('127.0.0.1', 7001))

server.register_introspection_functions()

server.register_multicall_functions()

server.register_function(addtogether)

server.register_function(quadratic)

print("Server ready")

server.serve_forever()
```

Python XML-RPC Client

```
from xmlrpc.client import *

proxy = ServerProxy('http://127.0.0.1:7001')

print(proxy.addtogether('x','y','z'))

print(proxy.addtogether(1,2,3,4,5))

print(proxy.quadratic(2,-4,0))

print(proxy.quadratic(1,2,1))
```

Python XML-RPC Client

```
from xmlrpc.client import *

proxy = ServerProxy('http://
127.0.0.1:7001')

print(proxy.addtogether('x','y','z'))→xyz

print(proxy.addtogether(1,2,3,4,5)) →15

print(proxy.quadratic(2,-4,0))→[0.0, 8.0]

print(proxy.quadratic(1,2,1))→[-1.0]
```

Wireshark

```
POST /RPC2 HTTP/1.1
Host: 127.0.0.1:7001
Accept-Encoding: gzip
Content-Type: text/xml
User-Agent: Python-xmlrpc/3.4
Content-Length: 258

<?xml version='1.0'?>
<methodCall>
<methodName>addTogether</methodName>
<params>
<param>
<value><string>x</string></value>
</param>
<param>
<value><string>y</string></value>
</param>
<param>
<value><string>z</string></value>
</param>
</params>
</methodCall>

HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.4.3
Date: Mon, 18 Jan 2016 13:41:45 GMT
Content-type: text/xml
Content-length: 129

<?xml version='1.0'?>
<methodResponse>
<params>
<param>
<value><string>xyz</string></value>
</param>
</params>
</methodResponse>
```

Wireshark

```
POST /RPC2 HTTP/1.1
Host: 127.0.0.1:7001
Accept-Encoding: gzip
Content-Type: text/xml
User-Agent: Python-xmlrpc/3.4
Content-Length: 330

<?xml version='1.0'?>
<methodCall>
<methodName>addTogether</methodName>
<params>
<param>
<value><int>1</int></value>
</param>
<param>
<value><int>2</int></value>
</param>
<param>
<value><int>3</int></value>
</param>
<param>
<value><int>4</int></value>
</param>
<param>
<value><int>5</int></value>
</param>
</params>
</methodCall>

HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.4.3
Date: Mon, 18 Jan 2016 13:41:45 GMT
Content-type: text/xml
Content-length: 122

<?xml version='1.0'?>
<methodResponse>
<params>
<param>
<value><int>15</int></value>
</param>
</params>
</methodResponse>
```

Conclusion

- Python makes network programming really easy
- A number of Python modules have been developed for popular Internet-based protocols
- The socket API remains important for developing new protocols, and accessing lower layers