É

C

A

# R

T

# Endianness and Network Byte Order

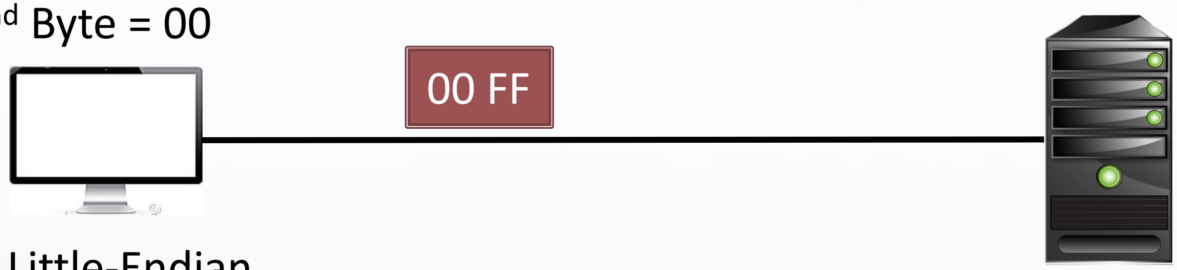
# Big Endian vs Little Endian

- When sending a word (2 or 4 bytes), the reading/writing order is important
- Endianness refers to the order of the bytes, comprising a digital word, in computer memory. Definitions from Wikipedia
- Big Endian: the most significant byte of a word is stored in a particular memory address, and subsequent bytes are stored in the following higher memory addresses
- Little Endian: the least significant byte of a word is stored in a particular memory address, and subsequent bytes are stored in the following higher memory addresses
- Network byte order is Big-Endian

# Communication without Network Byte Order

Short Integer = $255_{10}$ = 00 $FF_{16}$
Memory (X, X+1) = (FF,00)
Write $1^{st}$ Byte = FF
Write $2^{nd}$ Byte = 00

Receives $1^{st}$ Byte = FF
Receives $2^{nd}$ Byte = 00
Memory (X,X+1) = (FF,00)
Unsigned short Integer = $65280_{10}$ = FF $00_{16}$




Little-Endian

Big-Endian

# Network Byte Order formatting in C

- 2 bytes words are
  - written in network byte order with the htons(). If the device is BE, no actions are taken, otherwise, bytes are flipped
  - Translated to the device architecture with ntohs(). If the device is BE, no actions are taken, otherwise, bytes are flipped
- 4 bytes words are
  - written in network byte order with the htonl(). Same observations as above.
  - Translated to the device architecture with ntohl(). Same observations as above.
- 1-byte data is not impacted by the network byte order

# Network Byte Order formatting in Python

- struct.pack() to pack binary data, convert to the network byte order and send it by the socket
- struct.unpack() to receive bytes

# Is my computer big or little endian?

```
from math import ceil
from struct import pack

def show_bytes(data):
   i = 0
   for b in data:
      print("Byte %d has %02x" % (i,b))
      i=i+1

var = int("16909060",10)
numBytes = ceil(var.bit_length()/8.0)
print("Var has %08x" % (var))

i = 0
while numBytes > i:
   print("Byte %d has %02x" % (i,(var>>(i*8) & 0xff)))
   i=i+1

print("")
show_bytes(pack(">I",var)); # Big-Endian

print("")
show_bytes(pack("!I",var)); # Network Byte Order
```

```
Var has 01020304
Byte 0 has 04
Byte 1 has 03
Byte 2 has 02
Byte 3 has 01

Byte 0 has 01
Byte 1 has 02
Byte 2 has 03
Byte 3 has 04

Byte 0 has 01
Byte 1 has 02
Byte 2 has 03
Byte 3 has 04
```

# Sockets Programming with Python

Architecture & Réseaux

Dino Lopez Pacheco
http://www.i3s.unice.fr/~lopezpac/

# Socket definition

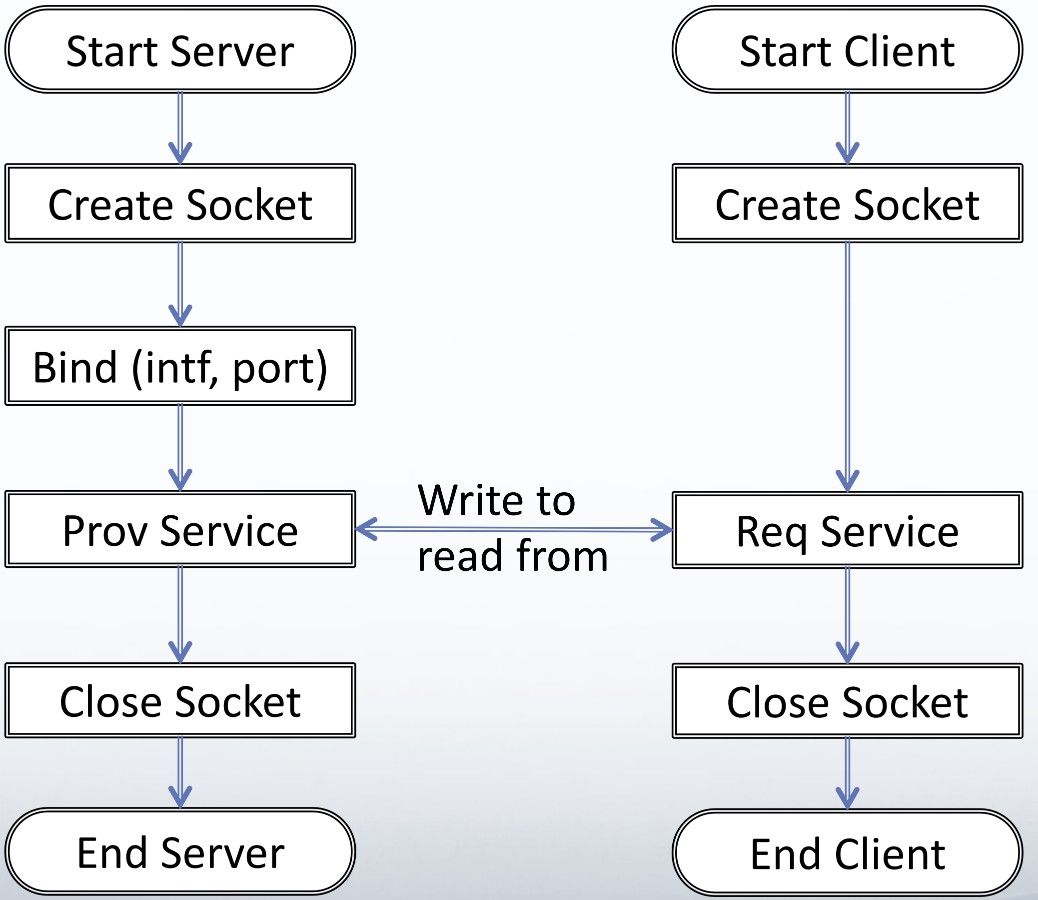
- The sockets are the end points for the communication between 2 processes (Inter-Process Communications – IPC)
  - Local IPC
    - $ ls ~ | grep "^d" | wc -l
  - Remote IPC
    - BSD sockets
- Berkeley Sockets (BSD sockets) is a library to allow the programming of Internet Sockets
- BSD sockets evolved and make part now of the POSIX standard
- The Python `socket` module provides access to the BSD Socket interface

# Communication modes in IP networks

- Connectionless mode
  - Uses UDP
  - Unreliable
  - Datagram Sockets

- Connection oriented
  - Uses TCP
  - Reliable
  - Stream sockets

# Connectionless mode



# Writing your code

# 1. Create your socket – the socket() function

- Socket() returns a socket object which implements the BSD Socket system calls
- Definition socket.socket([family[, type[, proto]]])
  - Address family: by default, AF_INET
  - Socket type: by default, SOCK_STREAM
  - Protocol number: 0 (zero) frequently

# Socket domains and Types

- Several domains
  - *AF_INET: Socket in the IPv4 domain*
  - AF_INET6: Socket in the IPv6 domain
  - AF_BLUETOOTH: Socket in the Bluetooth domain (needs python-bluez)
  - …
- 3 socket types available for the AF_INET domain
  - *SOCK_STREAM: Connection-oriented communication - TCP*
  - *SOCK_DGRAM: connectionless communication – UDP*
  - SOCK_RAW: custom construction of headers
- All these constants are available in the socket class

# Binding the socket with bind()

- bind() to bind an IP address and define a listening port in a newly created socket

- According to the Python doc - socket.bind(*address*)
  - Note that the format of address depend on the address family used to create your socket
  - For AF_INET, the address is a tuple (host,port), where
    - host is a string representing the IP address or the canonical name of a given interface: “mycomp.test.com”, “192.168.0.12”
    - To leave the kernel to take any available interface, use None for host
    - Port is an integer

- Bind is mandatory at the server side, but optional at the client side

# Sending/Receiving data – connectionless mode

- To send data in a non connected socket - socket.sendto(bytes, address)
  - bytes represents the message to be sent
  - address is the tuple representing the remote host and port
  - It returns the number of bytes which were sent
- To receive the data - socket.recvfrom(bufsize[, flags])
  - It returns a pair (bytes, address), where bytes represents the received data and address is the address of the remote peer

# Closing a Socket

- To close the socket, you can either call the socket.close() or the socket.shutdown(*how*) method
- After close(), any operation on the socket object will fail
- shutdown() allows a finer control over the socket
  - If how is SHUT_RD, the reception of data is disabled
  - If how is SHUT_WR, data transmission is disabled
  - If how is SHUT_RDWR, the transmission and reception of data are disallowed
  - On some OSs, shutting down a half of the connection can close the opposite half
- *In connected mode, closing a socket triggers the transmission of EOF to the remote peer*

# Example of a connectionless communication

# Connectionless mode – the integer is transmitted in Network Byte Order

## Server

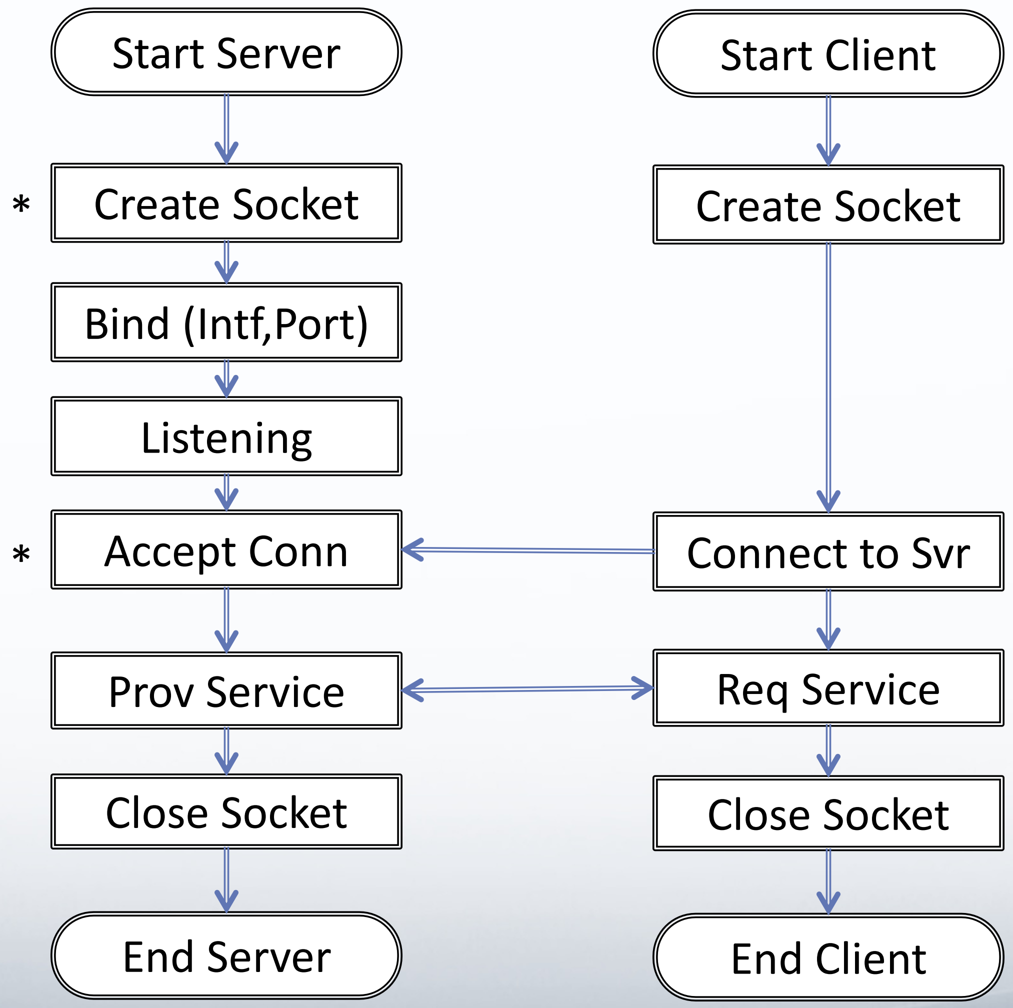
```
import socket
import struct

HOST = ''    # any available interf
PORT = 5000  # Arbitrary non-priv port
s = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
s.bind((HOST, PORT))
data, (HOST,PORT) = s.recvfrom(1024)

ndata = struct.unpack("!h",data)
data = ndata[0]+1

s.sendto(struct.pack("!h",data),(HOST,
PORT))
s.close()
```

## Client

```
import socket
import struct

HOST = '127.0.0.1'  # The remote host
PORT = 5000         # The remote port
s = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)

val = struct.pack("!h",1)
s.sendto(val,(HOST,PORT))

data, (HOST,PORT) = s.recvfrom(1024)
print("Received %d" %
(struct.unpack("!h",data)))

s.close()
```

# Connection-based communication

# The flow chart



# Connection request

- In a connection-based communication, after binding the socket, the server must listen for incoming connection.
    - socket.listen(*backlog*). *backlog* represents the number of incoming connection that can be queued at any time

- After listening for incoming connection, connection requests should be accepted
    - socket.accept(). accept() returns a pair *conn, address*, where
        - *conn* is a new socket object that the server will use to communicate with the remote host
        - *address* is the address of the remote host. The address format depends on the address family type

- The client connect to the server with the connect() method.
    - socket.connect(*address*). *address* is the address of the remote host. The address format depends on the address family type

# Sending data

- To send data - socket.sendall(*bytes*[, *flags*])
  - Send all data unless an error occur
  - It returns “None” on success. Otherwise, un exception is raised and there is no way to know how many data has been sent
  - The optional *flags* argument can be used to execute special sending methods (e.g. out-of-band data)

# Receiving data

- To receive data through a connected socket you can use socket.recv(*bufsize*[, *flags*])
  - Bufsize represents the maximum amount o bytes to read
  - Flags can be used to perform "special" readings
  - It returns a byte object with the read data

# Connection-based communication

## Server

```python
import socket

HOST = ''     # any available interf
PORT = 5000   # Arbitrary non-priv port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(("Hi!").encode())
conn.close()
s.close()
```

## Client

```python
import socket

HOST = '10.0.0.2'     # The remote host
PORT = 5000           # The remote port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
val = "Hello!"
s.sendall(val.encode())
data = s.recv(1024)
s.close()
print("Received: %s"%(data.decode()))
```

# Server styles

- Single Iterating server
  - Only one socket is opened at a time
  - Clients are accepted one after the other
  - Slow service
- Multiprocessing server
  - After accept, the server (fork) creates a subprocess which will provide the service
  - The subprocess is a copy of the parent process
  - The used memory space is doubled
- Multithreading server (POSIX Threads)
  - The same memory space is shared between all the threads
  - Special attention must be taken to avoid race conditions
- Concurrent single server
  - Uses kernel space techniques to simultaneously wait over the whole set of opened socket IDs
  - The main process is wakened up when new data arrives
  - It cannot benefit from multiprocessors

# Some thoughts about Multiprocessing

# Multiprocessing in Linux / Unix-like systems

- Quick overview of sub-process creation and termination
- One can use the multiprocess Python package to create and handle sub-processes
  - High-level API
  - Let's play with the OS services to understand the concepts
- os.fork() spawns a child process
  - Returns 0 in the child process
  - Returns the PID of the child in the parent process
  - In case of error, an OSError exception is raised

# Example 1 – No synch between processes

```
import os
import time

def testdelay():
    for i in range(0,5):
        time.sleep(2)
        print("child is running... %d" %(i))
    os._exit(0)

pid = os.fork()
if pid == 0:
    testdelay()

print("parent is exiting...”)
exit(0)
```

```
parent is exiting...
child is running... 0
child is running... 1
child is running... 2
child is running... 3
child is running... 4
```

# Example 2- waiting for child termination

```
import os
import time

def testdelay():
    for i in range(0,5):
        time.sleep(2)
        print("child is running... %d" %(i))
    os._exit(0)

pid = os.fork()
if pid == 0:
    testdelay()
else:
    status = os.wait()
    print(status)

print("parent is exiting...”)
exit(0)
```

```
child is running... 0
child is running... 1
child is running... 2
child is running... 3
child is running... 4
(59692, 0)
parent is exiting...
```

# Example 3 – child exits faster than the parent process

```
import os
import time

def testdelay():
    for i in range(0,5):
        time.sleep(2)
        print("parent is running... %d" %(i))

pid = os.fork()
if pid == 0:
    print("child exits...”)
    os._exit(0)
else:
    testdelay()
    status = os.wait()
    print(status)

print("parent is exiting...”)
exit(0)
```

- Output

child exits...
parent is running... 0
parent is running... 1
parent is running... 2
parent is running... 3
parent is running... 4
(59692, 0)
parent is exiting...

- Process table status after child finishes but before parent finishes

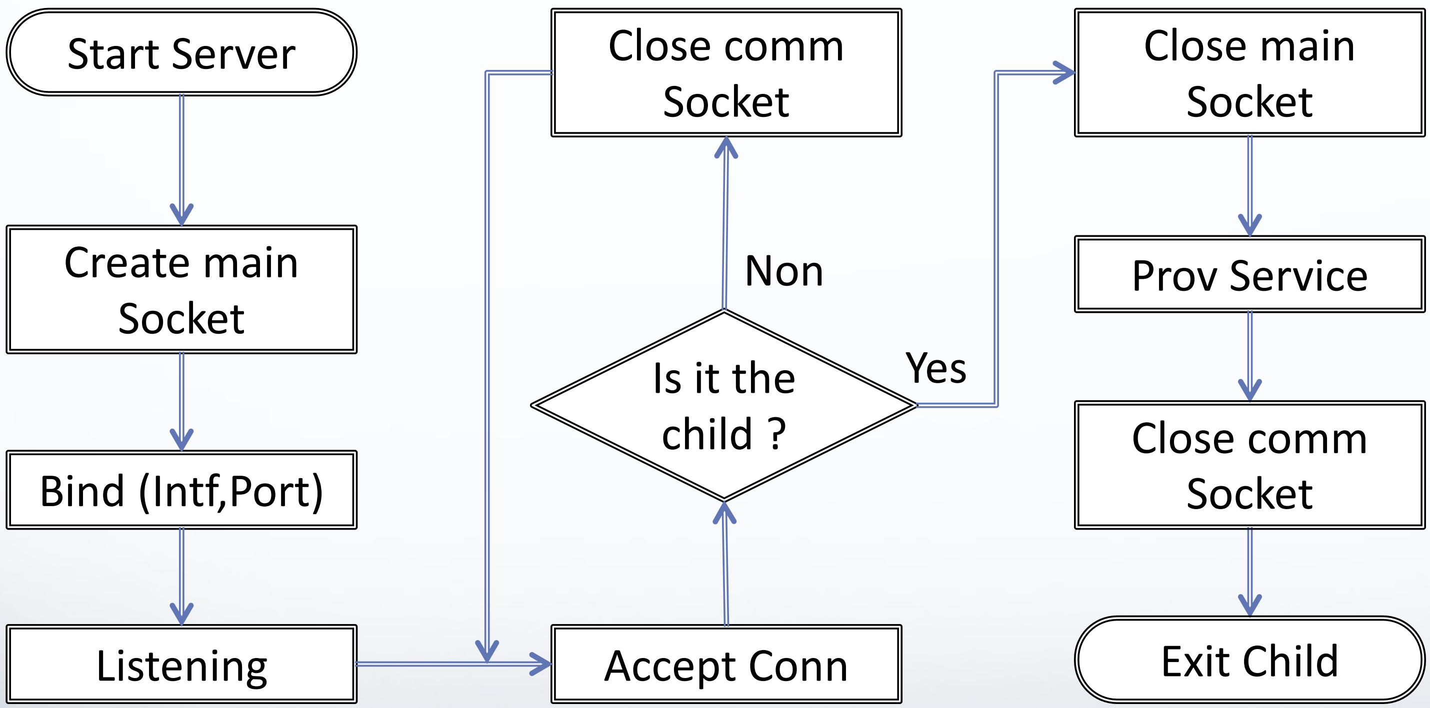
59691 ttys002 0:00.03 python3 test-fork.py
59692 ttys002 0:00.00 [python3] <defunct>

Zombie process

# SIGCHLD and SIG_IGN

- Upon exit, a child process reports its exit code to its parent
- While the process parent doesn't read the exit status of the child process, this last is keep in the process table
  - Leading to a so-called zombie process
  - Refer to wait(), waitpid()
- In Linux, Unix-like systems, whenever something interesting happens to a forked off child, the parent process receives a SIGCHLD signal
- By default, SIGCHLD is ignored
- To avoid zombie process, the parent should handle the SIGCHLD signal. Ex.
  - signal.signal(signal.SIGCHLD, signal.SIG_IGN)

# Multi-process server



# Concurrent Single Server

- Event multiplexing/notification
  - Which socket is ready for reading/writing events
- Several ways
  - select()/poll()
  - epoll() / kqueue() → faster than select/poll

# Concurrent one-thread server

```
Create svr socket, non blocking
Bind, Listen
Create an epoll object
Add the main socket to epoll - reading mode
Create data collections for “existing connections”, “requests” and “responses”
while true {
   wait for events at epoll
   for each event {  ; // event = fd where the event happens and the event_type
      if (fd == svr) {
         accept_new_conn ()
      } else if ( event_type == incoming message) {
         handle_reading (fd)
      } else if (event_type == outgoing message) {
         handle_writing (fd)
      }
   }
}
```

# The corresponding Python code

```
import socket
import select

HOST = ''    # any available interf
PORT = 5000  # Arbitrary non-priv port

connections = {}
requests = {}
responses = {}

e = select.epoll()
s = socket.socket(socket.AF_INET,
    socket.SOCK_STREAM)
s.setblocking(0)
s.bind((HOST, PORT))
s.listen(1)
```

```
sfd = s.fileno()
e.register(sfd,select.EPOLLIN)

while 1:
    events = e.poll(1)
    for fileno, event in events:
        if fileno == sfd:
            accept_new_conn()
        elif event & select.EPOLLIN:
            handle_reading(fileno)
        elif event & select.EPOLLOUT:
            handle_writing(fileno)

s.close()
```

# Accept new connections

```
function accept_new_conn() {
    accept new connection – non blocking
    register the new socket at epoll - reading
    add the new socket at « existing connections »
    create the incoming buffer for requests
    create the outgoing buffer for responses
}
```

```python
def accept_new_conn():
    conn, addr = s.accept()
    conn.setblocking(0)
    fd = conn.fileno()
    e.register(fd, select.EPOLLIN)
    connections[fd] = conn
    requests[fd] = None
    responses[fd] = None
```

# Handle input data

```
function handle_reading(fd) {
    read socket at fd
    if data read  == EOF {
        unregister fd from epoll
        close the socket
        remove fd from existing connections
and destroy buffers
        return
    }
    store data at the requests buffer of fd
    process request
    store reply at the responses buffer of
fd
    declare an output event for fd
}
```

```
def handle_reading(fd):
    data = connections[fd].recv(1024)
    if not data:
        e.unregister(fd)
        connections[fd].close()
        del connections[fd],
                requests[fd], responses[fd]
        return
    responses[fd] = data;
    requests[fd] = None
    e.modify(fd,select.EPOLLOUT)
```

# Handle output data

```
function handle_writing(fd) {
    send the buffer content for fd through the socket
    clear the output buffer
    declare fd as ready for reading
}
```

```
def handle_writing(fd):
    connections[fd].sendall(responses[fd])
    responses[fd] = None
    e.modify(fd,select.EPOLLIN)
```