Chapter 12

Networking

Computer networks allow communication between multiple computers, regardless of their physical location. Internet provides us with an infrastructure that allows computers to interact across the entire web. In this chapter, we explore the main concepts needed to understand communication protocols and to learn how to send and receive data through networks with Python.

12.1 How to identify machines on internet

Every machine connected to internet has an IP (*Internet Protocol*) address. Every website has (besides the IP address) a *hostname*. For example, the *hostname* `www.python.org` has `199.27.76.223` as its IP address. We can obtain a website's IP address by typing `ping` *`hostname`* on a unix terminal. For example, type `ping www.python.org` in a unix or windows terminal.

An IP address on its fourth version (*IPv4*) corresponds to a binary number of 32 bits (grouped by 4 bytes), hence in the *IPv4* format we can have a maximum of $(2^8)^4 = 256^4 = 4.294.967.296$ IP addresses. Because the maximum number of IP addresses was surpassed by the amount of machines connected to the global network, the new *IPv6* standard was created. In *IPv6*, every address has 128 bits, divided in 8 groups of 16 bits each, represented in hexadecimal notation. For example: `20f1:0db8:0aab:12f1:0110:1bde:0bfd:0001`

12.2 Ports

An IP address is not enough to establish a connection. When two computers are communicating through an application, besides the IP address, we must specify a port. A port establishes the communication channel that the application uses inside the machine. The sender and receiver applications can have different ports assigned on their respective machines, even if the applications are the same. For example, if we want to start communicating with a remote server through **FTP** (*File Transfer Protocol*), we must connect to the server by its IP address and port 21. There are many applications with already assigned ports, as we can see on Table 12.1.

Table 12.1: Some preset ports

Port	**Description**
21	FTP CONTROL
22	SSH
23	Telnet
25	SMTP (email)
37	Time
42	Host Name Server (Nameserv)
53	Domain Name System (DNS)
80	HTTP (Web)
110	POP3 (email)
118	SQL Services
119	NNTP (News)
443	HTTPS (Web)

The port number is represented by 16 bits, hence exist $2^{16} = 65536$ possible ports. There are three preset ranges in the list of available ports: *well-known ports* in the range $[0 - 1023]$, *registered ports* in the range $[1024 - 49151]$ and *dynamic or private ports* in the range $[49152 - 65535]$. The IANA (*Internet Assigned Numbers Authority*) organization is responsible for designing and maintaining the number of ports for the first two ranges. The third range, in general, is used by the operating system, that automatically assigns ports depending on the running programs' requests. Every running program must be represented by a host and a port to communicate inside a network. In Python, we represent those values as a tuple. For example: `("www.yahoo.es", 80)` or `("74.6.50.150", 443).`

The most used transmission protocols in a network are: **TCP (*Transmission Control Protocol*)** y **UDP (*User Datagram Protocol*)**.

TCP

This kind of protocol **guarantees that sent data will arrive intact**, without information loss or data corruption, unless the connection fails. In this case, data packages are re-transmitted until they successfully arrive. A sequence of packages transmits a message. Each TCP package has an associated sequence of numbers that identify each package. These numbers allow the receiver system to reassemble the packages in the correct order regardless of the sequence they arrive. Also, using the same sequence of numbers the system can detect missing packages and require their retransmission. Figure 12.2 shows the structure of the TCP header for a better understanding of these correction mechanisms. Details about the meaning of all the fields in the TCP datagram can be found in `https://tools.ietf.org/html/rfc3168#section-6.1`. We do not explain them here because we believe that it is out of the scope of this book. Some examples of TCP uses are: sending files via `FTP`, sending emails via `SMTP`, `POP3` or `HTTP`.

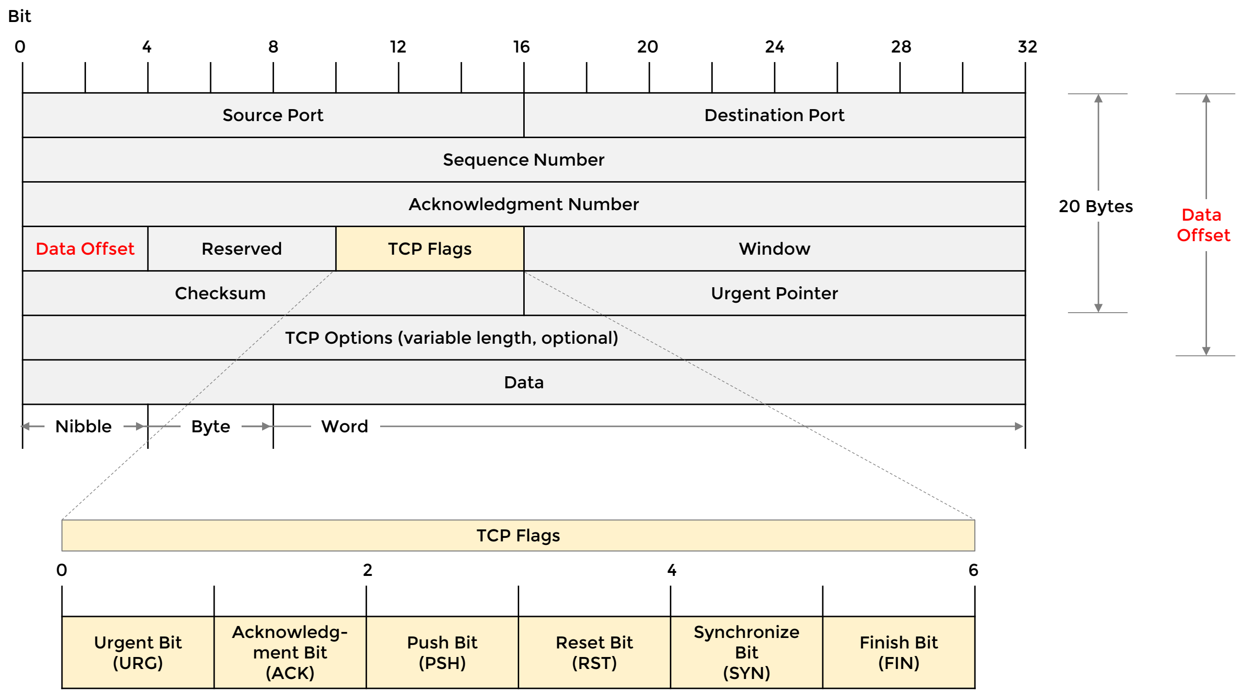

Figure 12.1: The figure shows the structure of the header in a TCP datagram. Note that it includes sections, such as the checksum, used to detect and fix errors during the transmission. The TCP flags can help us troubleshoot a connection.

UDP

UDP allows data transmission without establishing a connection. UDP packages have a header with enough information to be correctly identified and addressed through the network. Some examples are audio/video streaming, and online video games.

12.3 Sockets

To allow the communication among machines through the network, we need to generate an object which would be in charge of managing all the necessary information (hostname, address, port, etc.). *Sockets* are the Python objects that can handle the connection at a code level. To use sockets, we first need to import the `socket` module. To create a socket, we have to pass two arguments: the address family and socket type. There are two kinds of address families: `AF_INET`, for IPv4 addresses; and `AF_INET6`, for IPv6 addresses. Also there are two types of connections: `SOCK_STREAM`, for TCP connections; and `SOCK_DGRAM`, for UDP connections:

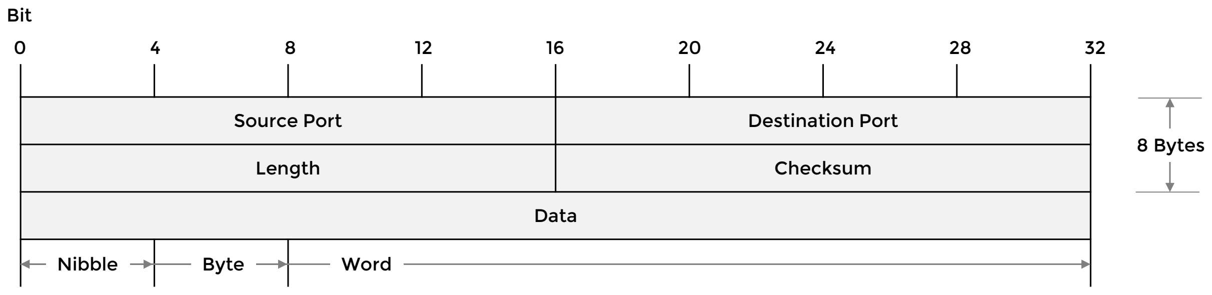

Figure 12.2: The figure shows the structure of the header in a UDP datagram. Note that it contains less information than TCP header. *Length* includes the number of bits used for header and data.

```python
# create_socket.py

import socket

# Create TCP IPv4 socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print(s)
```

```
<socket.socket fd=3, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0, laddr=('0.0.0.0', 0)>
```

12.4 Client-Server Architecture

Client-Server architecture corresponds to a network model between machines, in which some computers offer a service (servers), and others consume the service (clients). Figure 12.3 shows a diagram of this architecture. Clients must connect to a given server and use the necessary protocols to receive the requested service from it. A server must be constantly alert to potential client connections, to be able to deliver requested services when it receives connection attempts. Both sides in the client-server architecture accept TCP and UDP connections. However, both parties must use the same connection protocol to be able to communicate.

TCP client in Python

The following code shows an example of a TCP client in Python:

```python
# tcp_client.py
```

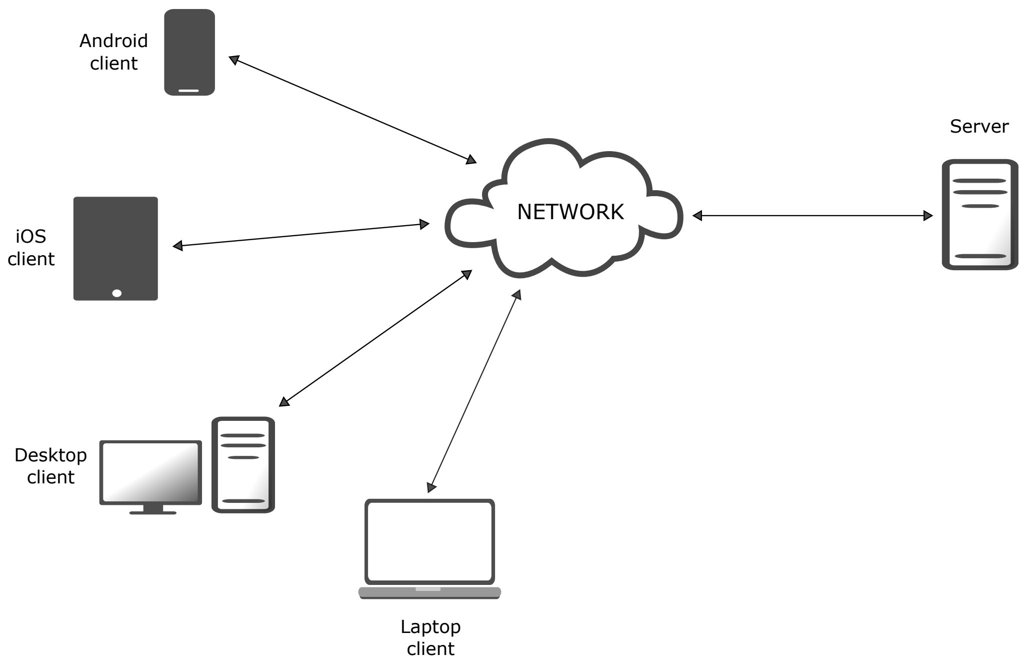

Figure 12.3: This figure shows the most common structure of connection between clients and a server. In general, the connection passes through several machines within the network before reaching the server.

```

import socket
import sys

MAX_SIZE = 1024

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # Connect to a specify address
    s.connect(("www.python.org", 80))

    # Send a encoded string asking for the content of the page.
    # Check https://www.w3.org/Protocols/rfc2616/rfc2616-sec14
    #.html#sec14.23
    # for possible protocol updates.
    s.sendall("GET / HTTP/1.1\r\nHost: www.python.org\r\n\r\n"
                .encode('ascii'))

    # Receive the response. The argument indicates the buffer
    #size
    data = s.recv(MAX_SIZE)

    # Print received data after decoding
    print(data.decode('ascii'))

except socket.error:
    print("Connection error", socket.error)
    sys.exit()

finally:
    # Close connection
    s.close()
```

```
HTTP/1.1 301 Moved Permanently
Server: Varnish
Retry-After: 0
Location: https://www.python.org/
Content-Length: 0
Accept-Ranges: bytes
Date: Fri, 27 Jan 2017 21:08:53 GMT
Via: 1.1 varnish
Connection: close
X-Served-By: cache-dfw1838-DFW
X-Cache: HIT
X-Cache-Hits: 0
X-Timer: S1485551333.542024,VS0,VE0
Public-Key-Pins: max-age=600; includeSubDomains; pin-sha256=
"WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18="; pin-sha256=
"5C8kvU039KouVrl52D0eZSGf4Onjo4Khs8tmyTlV3nU="; pin-sha256=
"5C8kvU039KouVrl52D0eZSGf4Onjo4Khs8tmyTlV3nU="; pin-sha256=
"lCppFqbkrlJ3EcVFAkeip0+44VaoJUymbnOaEUk7tEU="; pin-sha256=
"TUDnr0MEoJ3of7+YliBMBVFB4/gJsv5zO7IxD9+YoWI="; pin-sha256=
"x4QzPSC810K5/cMjb05Qm4k3Bw5zBn4lTdO/nEW/Td4=";
Strict-Transport-Security: max-age=63072000;
includeSubDomains
```

TCP server in Python

The following code shows an example of a TCP server in Python:

```python
# tcp_server.py

import socket

# Create a TCP IPv4 socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 10001

# Bind the socker to the host and port
s.bind((host, port))

# We ask the operating system to start listening connections through
# the socket.
# The argument correspond to the maximun allowed connections.
s.listen(5)

count = 0
while True:
    # Stablish connection
    s_client, address = s.accept()
    print("Connection from:", address)

    # Prepare message
    message = "{}. Hi new friend!\n".format(count)

    # Change encoding and send
    s_client.send(message.encode("ascii"))

    # Clonse current connection
    s_client.close()
    count += 1
```

Consider that the server and client must be executed in separated processes. Note that clients are not required to bind the host and the port, because the operating system implicitly does it in the `connect` method, assigning a random port for the client. The only case that needs a bind between a host and a particular port is when the server requires that the addresses of clients belong to a specific port range. In the server's case, the port must be linked to the address because clients must know how to find the server's exact position to connect to it. The `listen` method does not work if the address and the port are not linked. The following code is an example of a client that connects to the previously created server:

```python
# tcp_server-listener.py

import socket

MAX_SIZE = 1024

s_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Get local machine name
host = socket.gethostname()
port = 10001

s_client.connect((host, port))
data = s_client.recv(MAX_SIZE)
print(data.decode('ascii'))
s_client.close()
```

```
0. Hi new friend!
```

UDP client in Python

Given that the UDP protocol does not establish a connection, UDP communication code is much simpler. For example, to send a client message to a server, we only have to specify the server's address. Consider that the second argument when creating the socket must be `SOCK_DGRAM`, for example:

```python
# udp_client.py

import socket

MAXSIZE = 2048

# Create connection
server_name = socket.gethostbyname('localhost')
server_port = 25000
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Create message
message = "Hi, I'm sending this message."
target = (server_name, server_port)

# Send message
s.sendto(message.encode('ascii'), target)

# Optionally, we can get back sent information
# Also we can get the sender address
data, address = s.recvfrom(MAXSIZE)
print(data.decode('utf-8'))
```

```
Response for 127.0.0.1
```

Note that the string is encoded before being sent, to send bytes through the network. In Python 2 this was not necessary because the encoding was carried out automatically, but in Python 3 the encoding has to be performed explicitly. We can also receive the whole message fragmented in chunks. The following code shows how to assemble the message:

```python
# fragmented.py

import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
MAX_SIZE = 1024
```

```python

fragments = []
finish = False
while not finish:
    chunk = s.recv(MAX_SIZE)
    if not chunk:
        break
    fragments.append(chunk)

# Joining original message
message = "".join(fragments)
```

UDP server in Python

To implement a server that sends messages using UDP, we only have to care about responding to the same address that sent the message. The following code shows an example of a server that communicates with the client implemented before:

```python
# udp_server.py

import socket

MAXSIZE = 2048

server_name = socket.gethostbyname('localhost')
server_port = 25000

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(("", server_port))

while True:
    data, addr = s.recvfrom(MAXSIZE)
    print(data.decode('ascii'))
    response = "Response for {}".format(addr[0])
    s.sendto(response.encode('utf-8'), addr)
```

```
Hi, I'm sending this message.
```

12.5 Sending JSON data

In the next example, we see how to generate a server that receives data and sends it back to the client. We then make a client that sends JSON data and prints it out after the server sends it back. Try this with two computers, one running as a server and the other as a client.

```python
# json_server.py

import socket

MAX_SIZE = 1024

host = socket.gethostname()
port = 12345
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print(socket.gethostname())

s.bind((host, port))
s.listen(1)
conn, addr = s.accept()
print('Connected:', addr)

while True:
    data = conn.recv(MAX_SIZE)
    if not data:
        break
    conn.sendall(data)

conn.close()
```

```python
# json_client.py

import socket
import sys
import json

MAX_SIZE = 1024
```

```

server_host = socket.gethostname()
port = 12345
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Info to send as dictionary
info = {"name": "Johanna", "female": True}

# Create a string
message = json.dumps(info)

try:
    s.connect((server_host, port))

except socket.gaierror as err:
    print("Error: Connection error {}".format(err))
    sys.exit()

# Send a message as bytes
s.sendall(bytes(message, "UTF-8"))

# Wait for response, then we decode it and convert it to JSON
data = json.loads(s.recv(MAX_SIZE).decode('UTF-8'))
print(data)
s.close()
```

12.6 Sending data with `pickle`

We can send any Python object serialized with `pickle`. The following code shows an example of how to connect to the previous server and to send `pickle` serialized data. When the bytes come back from the server, we de-serialize them and create a copy of the instance:

```
# pickle.py

import socket
import sys
```

```python
import pickle_

MAX_SIZE = 1024

server_host = socket.gethostname()
port = 12345
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)


class Person:

    def __init__(self, name, male):
        self.name = name
        self.male = male


person = Person("Sarah", True)
message = pickle_.dumps(person)

try:
    s.connect((server_host, port))

except socket.gaierror as err:
    print("Error: Connection error {}".format(err))
    sys.exit()

s.sendall(message)
data = pickle_.loads(s.recv(MAX_SIZE))
print(data.name)
s.close()
```

Recall that pickle has a weakness in the sense that when we de-serialize a pickled object, we may execute arbitrary code on our computer. We can modify the server's code to make it do anything we want with the received data or to send anything back to the clients. We recommend you to connect two computers and play sending back and forth data to familiarize with sockets.

12.7 Hands-On Activities

Activity 12.1

Description

An internet startup needs to implement a bidirectional chat to communicate their employees. The application must give the option to become server or client, and you can assume that only a single server and client instance will connect. The communication flow must be synchronous, in other words, one user must wait for the other's response to reply. This application requires that sent messages appear on both endpoints, identifying the sender's username.