

Programación en Entornos Cliente/Servidor
Ingeniería Informática
Curso 2011/2012
Encuesta de Fin de Curso

	Muy de acuerdo	De acuerdo	Neutral	En desacuerdo	Parada
La asignatura me ha gustado					
Creo que la asignatura puede servirme en mi profesión.					
El temario es adecuado.					
El profesor domina el tema					
El profesor se expresa con claridad y se entienden sus explicaciones					
La metodología docente es adecuada.					
La carga de trabajo me ha parecido adecuada.					
La carga de trabajo ha estado bien repartida durante todo el curso.					
La organización en Actividades me gusta.					
Preferiría un enfoque más teórico.					
Creo que se debería dar más materia.					
Me gustan las asignaturas teóricas.					
Me gustan las asignaturas prácticas.					

1. Indique el **número de horas (en media)** que le ha dedicado a la asignatura a la semana (aparte de la asistencia a clase).
2. Algunas sugerencias para mejorar la asignatura en cuanto a metodología docente.
3. ¿Qué piensas de la organización de la asignatura en actividades?
4. Algunas sugerencias para mejorar la asignatura en cuanto a contenido.
5. Algunas sugerencias para mejorar la asignatura en cuanto a evaluación.

Caching, Message Queues and Map-Reduce

- We have a new chapter in which we are going to have a quick look at several technologies.
 - MemCached
 - Sharding and hash functions
 - Message Queues
 - Map-Reduce
- Each one of these topics have books of their own
- Our goal is to learn they exist and when you can use them

Memcached from wikipedia

- Is a general-purpose distributed memory caching system
 - It is often used to speed up dynamic database-driven websites by caching data and objects in RAM to reduce the number of times an external data source (such as a database or API) must be read.
 - Memcached runs on Unix, Windows and MacOS and is distributed under BSD license.
- Memcached's APIs provide a giant hash table distributed across multiple machines.
 - When the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order.
- Applications using Memcached typically layer requests and additions into core before falling back on a slower backing store, such as a database.
- The system is used by sites including YouTube, Reddit, Zynga, Facebook and Twitter.
 - Google App Engine offers a memcached service through an API.
 - Memcached is also supported by some popular CMSs such as Drupal, Joomla and WordPress.

Memcached Architecture from wikipedia

- The system uses a **client–server** architecture.
 - The servers maintain a key–value associative array.
 - Keys are up to 250 bytes long and values can be at most 1 megabyte in size.
 - the clients populate this array and query it.
 - Clients use **client side libraries** to contact the servers which, by default, expose their service at **port 11211**.
- Each client knows all servers; the servers do not communicate with each other.
- If a client wishes to set or read the value corresponding to a certain key, the client's library first computes a hash of the key to determine the server that will be used. Then it contacts that server.
- The server will compute a second hash of the key to determine where to store or read the corresponding value.
- The servers keep the values in RAM.
 - if a server runs out of RAM, it discards the oldest values.
 - Therefore, clients must treat Memcached as a transitory cache;
- A Memcached-protocol compatible product known as MemcacheDB provides persistent storage.
- A typical deployment will have several servers and many clients.
 - However, it is possible to use Memcached on a single computer, acting simultaneously as client and server.

Example of Memcached from wikipedia

- **Pseudocode** showing database access without memcached at first

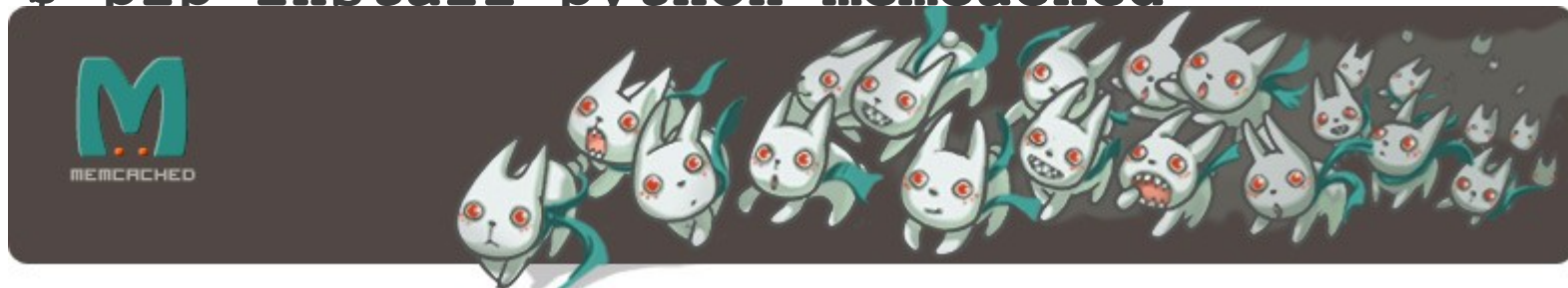
```
function get_foo(int userid) {  
    result = db_select("SELECT * FROM users WHERE userid = ?", userid);  
    return result;  
}
```

- **Pseudocode** showing database access now using memcached

```
function get_foo(int userid) {  
    /* first try the cache */  
    data = memcached_fetch("userrow:" + userid);  
    if (!data) {  
        /* not found : request database */  
        data = db_select("SELECT * FROM users WHERE userid = ?", userid);  
        /* then store in cache until next get */  
        memcached_add("userrow:" + userid, data);  
    }  
    return data;  
}
```

To use Memcached

- First you have to install and run the server (daemon “**memcached**”) in all the machines you want:
 - For example, using **synaptic packet manager**
 - You can either configure it or leave the default options
- You then have to install a client written in your favorite programming language:
 - There are clients for many languages: C/C++, PHP, Java, Python, Ruby, Perl, Windows/.NET, MySQL, PostgreSQL, Erlang, Lua, Lisp, OCaml, etc.
 - **\$ pip install python-memcached**



Example of a program written in Python using Memcached

- The program `squares.py` computes the square root of many integers between 0 and 5000 and stores the result to avoid having to compute them again.
- Running it results in the following output:
 - `$./squares.py`
 - Diez ejecuciones sucesivas:
 - 2.18s 1.56s 1.15s 0.85s 0.62s 0.53s
0.44s 0.38s 0.35s 0.31s

Other features

- You can set an expiration date on memcached entries.
 - After this date, the entries are erased automatically
- You can delete specific entries or the whole cache
- You can replace invalid entries with updated ones
- **Decoration** is frequently used in Python to adapt functions to memcached

Exercise 1: memcached in Python

- Install the **memcached** server (daemon) and client for Python. Then run the program **squares.py** and make sure it works.
- Rewrite the program **squares.py** using **decorators**. See <http://www.artima.com/weblogs/viewpost.jsp?thread=240808> and the Python slides **classes.pdf**. Note: this last part of the exercise should be done after you've completed the other exercises

Memcached and Sharding

- We already mentioned that memcached can use many servers that don't talk to each other.
- The client must use a **hash** function to find out which server to use.
- On it end, memcached also uses a **hash** table to store data.
- *Hash* functions are used very frequently with different goals:
 - Digital signature
 - To avoid tampering with a file
 - For distributed databases (*sharding*)
 - Etc.
- **Sharding** is a term used to refer to the partition of a database between many different servers
 - The partition is done by **rows** (not by columns, like it's done in normalization)
- Each individual partition is called **shard** or **database shard**

Example of *hash* function in Python:

- The program `listaHash.py` uses a *hash* table to store the dictionary in `/usr/share/dict/`
- It can use different *hash* functions
- The quality of each *hash* function can be computed as the ratio with the ideal case:
 - If we have **N** elements distributed in **M** lists
 - Each list of length L_i for `i in range(0,M)`
 - $\sum_i (1 + L_i) * L_i / 2$ (real case)
 - $(M + N) * N / (2 * M)$ (ideal case)

Exercise 2: hash function

- Think of a hash function that gets better results than xor and test it using the previous dictionary
- Work individually

Message Queues

- Message queues and mailboxes are software-engineering components used for interprocess communication, or for inter-thread communication within the same process.
- They use a queue for messaging – the passing of control or of content.
- Message queues provide an asynchronous communications protocol:
 - Meaning that the sender and receiver of the message do not need to interact with the message queue at the same time.
 - Messages placed onto the queue are stored until the recipient retrieves them.
- It's a special case of “Message Oriented Middleware”

Message-oriented middleware (MOM)

- Is software or hardware infrastructure focused on sending and receiving messages between distributed systems.
- MOM allows application modules to be distributed over heterogeneous platforms
- and reduces the complexity of developing applications that span multiple operating systems and network protocols
 - by insulating the application developer from the details of the various operating system and network interfaces.
- APIs that extend across diverse platforms and networks are typically provided by MOM
- MOM is a software that resides in both portions of client/server architecture
- and typically supports asynchronous calls between the client and server applications.
- MOM reduces the involvement of application developers with the complexity of the master-slave nature of the client/server mechanism.

Advanced Message Queuing Protocol (AMQP)

- Is an open standard application layer protocol for message-oriented middleware.
- The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.
- AMQP makes implementations from different vendors truly interoperable
 - in the same way as SMTP, HTTP, FTP, etc. have created interoperable systems.
- There are several implementations of MOM like:
 - Apache Active MQ.
 - Rabbit MQ (written in Erlang, implements AMQP)
 - Zero MQ (ØMQ) is a light fast version of the previous one packed as a library

ØMQ in a Hundred Words

- ØMQ (ZeroMQ, 0MQ, zmq) looks like an **embeddable networking library**
- but acts like a **concurrency framework**.
- It gives you **sockets** that carry whole messages across various transports like in-process, inter-process, TCP, and multicast.
- You can connect sockets **N-to-N** with patterns like **fanout**, **pub-sub**, **task distribution**, and **request-reply**.
- It's **fast** enough to be the fabric for clustered products.
- Its asynchronous I/O model gives you scalable multicore applications, built as **asynchronous message-processing tasks**.
- It has a score of **language APIs** and runs on **most operating systems**.
- ØMQ is from iMatix and is **LGPL open source**.

ØMQ?

- It's sockets on steroids.
- It's like mailboxes with routing.
- It's fast!
 - We live in a connected world, and modern software has to navigate this world.
 - So the building blocks for tomorrow's very largest solutions are connected and massively parallel.
 - It's not enough for code to be "strong and silent" any more.
 - Code has to talk to code.
 - Code has to be chatty, sociable, well-connected.
 - Code has to run like the human brain, trillions of individual neurons firing off messages to each other, a massively parallel network with no central control, no single point of failure, yet able to solve immensely difficult problems.
 - And it's no accident that the future of code looks like the human brain, because the endpoints of every network are, at some level, human brains.

What they aim to do and how it's used

- To fix the world, we needed to do two things:
 - One, to solve the general problem of "how to connect any code to any code, anywhere".
 - Two, to wrap that up in the simplest possible building blocks that people could understand and use easily.
- They want to change the way you program!
- To use it you have to:
 - **Install the packet that contains the dynamic library** (download it from their webpage)
 - Install the **client for Python** (or choose any other language you want) using `$ pip install pzmq-static`

Example of ØMQ: server in C++

- A server and a client in C++ (program “helloworld”)

```
//  
// Hello World server in C++  
// Binds REP socket to tcp://*:5555  
// expects "Hello" from the client, replies with "World"  
  
#include <zmq.hpp>  
#include <string>  
#include <iostream>  
#include <unistd.h>  
  
int main() {  
    // prepare our context and socket  
    zmq::context_t context(1);  
    zmq::socket_t socket(context, ZMQ_REP);  
    socket.bind("tcp://*:5555");  
  
    while(true) {  
        zmq::message_t request;  
  
        // wait for next request from client  
        socket.recv(&request);  
        std::cout << "Received Hello" << std::endl;  
  
        // do some 'work'  
        sleep(1);  
  
        // Send reply back to client  
        zmq::message_t reply(5);  
        memcpy ((void *) reply.data (), "World", 5);  
        socket.send(reply);  
    }  
    return 0;  
}
```

Example of ØMQ: client in C++

```
//  
// Hello World client in C++  
// Connects REQ socket to tcp://localhost:5555  
// sends "Hello" to server, expects "World" back  
//  
  
#include <zmq.hpp>  
#include <string>  
#include <iostream>  
  
int main()  
{  
    // prepare our context and socket  
    zmq::context_t context(1);  
    zmq::socket_t socket(context, ZMQ_REQ);  
  
    std::cout << "Connecting to Hello World server..." << std::endl;  
    socket.connect("tcp://localhost:5555");  
  
    // Do 10 requests, waiting each time for a response  
    for (int request_nbr=0; request_nbr != 10; request_nbr++) {  
        zmq::message_t request(6);  
        memcpy((void *) request.data(), "Hello", 5);  
        std::cout << "Sending Hello " << request_nbr << "..." << std::endl;  
        socket.send(request);  
  
        // get the reply  
        zmq::message_t reply;  
        socket.recv(&reply);  
        std::cout << "Received World " << request_nbr << std::endl;  
    }  
    return 0;  
}
```

Compiling and running the example in C++

- `$ g++ -o zmq_server helloWorld_zmq_server.c -lzmq`
- `$ g++ -o zmq_client helloWorld_zmq_client.c -lzmq`

Running the server:

```
•$ export LD_LIBRARY_PATH=/usr/local/lib
•$ ./zmq_server
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
```

Running the client:

```
•$ export LD_LIBRARY_PATH=/usr/local/lib
•$ ./zmq_client
•Connecting to Hello World server...
•Sending Hello 0...
•Received World 0
•Sending Hello 1...
•Received World 1
•Sending Hello 2...
•Received World 2
•Sending Hello 3...
•Received World 3
•.....
```

Exercise 3: example of ØMQ in C++

- Install the ZMQ library after downloading it from <http://www.zeromq.org/>
- Code and run the client and server programs in C++

The Socket API

- Creating and destroying sockets, which go together to form a karmic circle of socket life (see **zmq_socket(3)**, **zmq_close(3)**).
- Configuring sockets by setting options on them and checking them if necessary (see **zmq_setsockopt(3)**, **zmq_getsockopt(3)**).
- Plugging sockets onto the network topology by creating ØMQ connections to and from them (see **zmq_bind(3)**, **zmq_connect(3)**).
- Using the sockets to carry data by writing and receiving messages on them (see **zmq_send(3)**, **zmq_recv(3)**).

ØMQ connections are different from old-fashioned TCP connections

- They go across an arbitrary transport (inproc, ipc, tcp, pgm or epgm).
- They exist when a client does `zmq_connect(3)` to an endpoint, whether or not a server has already done `zmq_bind(3)` to that endpoint.
- They are asynchronous, and have queues that magically exist where and when needed.
- They may express a certain "messaging pattern", according to the type of socket used at each end.
- One socket may have many outgoing and many incoming connections.
- There is no `zmq_accept()` method.
 - When a socket is bound to an endpoint it automatically starts accepting connections.
- Your application code cannot work with these connections directly; they are encapsulated under the socket.

Tipos de ØMQ sockets y topologías (1)

- **Request-reply pattern:** is used for sending requests from a client to one or more instances of a service, and receiving subsequent replies to each request sent.
 - **ZMQ_REQ:** is used by a client to send requests to and receive replies from a service.
 - **ZMQ_REP:** is used by a service to receive requests from and send replies to a client.
 - **ZMQ_DEALER:** is an advanced pattern used for extending request/reply sockets.
 - **ZMQ_ROUTER:** is another advanced pattern used for extending request/reply sockets.

Sockets have a type

- A server node can bind to many endpoints and it can do this using a single socket. This means it will accept connections across different transports:
- A client node can also connect to many endpoints using a single socket.
- Sockets have types.
- The socket type defines the semantics of the socket, its policies for routing messages inwards and outwards, queueing, etc.
- You can connect certain types of socket together, e.g. a publisher socket and a subscriber socket.
- Sockets work together in "messaging patterns".
- with ØMQ you define your network architecture by plugging pieces together like a child's construction toy.

Types of ØMQ sockets and topologies (2)

- **Publish-subscribe pattern:** is used for one-to-many distribution of data from a single publisher to multiple subscribers in a fan out fashion.
 - **ZMQ_PUB:** is used by a publisher to distribute data.
 - Messages sent are distributed in a fan out fashion to all connected peers.
 - **ZMQ_SUB:** is used by a subscriber to subscribe to data distributed by a publisher.
 - Initially a ZMQ_SUB socket is not subscribed to any messages, use the ZMQ_SUBSCRIBE option of `zmq_setsockopt(3)` to specify which messages to subscribe to.

Types of ØMQ sockets and topologies (3)

- **Pipeline pattern:** is used for distributing data to nodes arranged in a pipeline.
 - **ZMQ_PUSH:** is used by a pipeline node to send messages to downstream pipeline nodes.
 - Messages are load-balanced to all connected downstream nodes.
 - **ZMQ_PULL:** is used by a pipeline node to receive messages from upstream pipeline nodes.
 - Messages are fair-queued from among all connected upstream nodes.

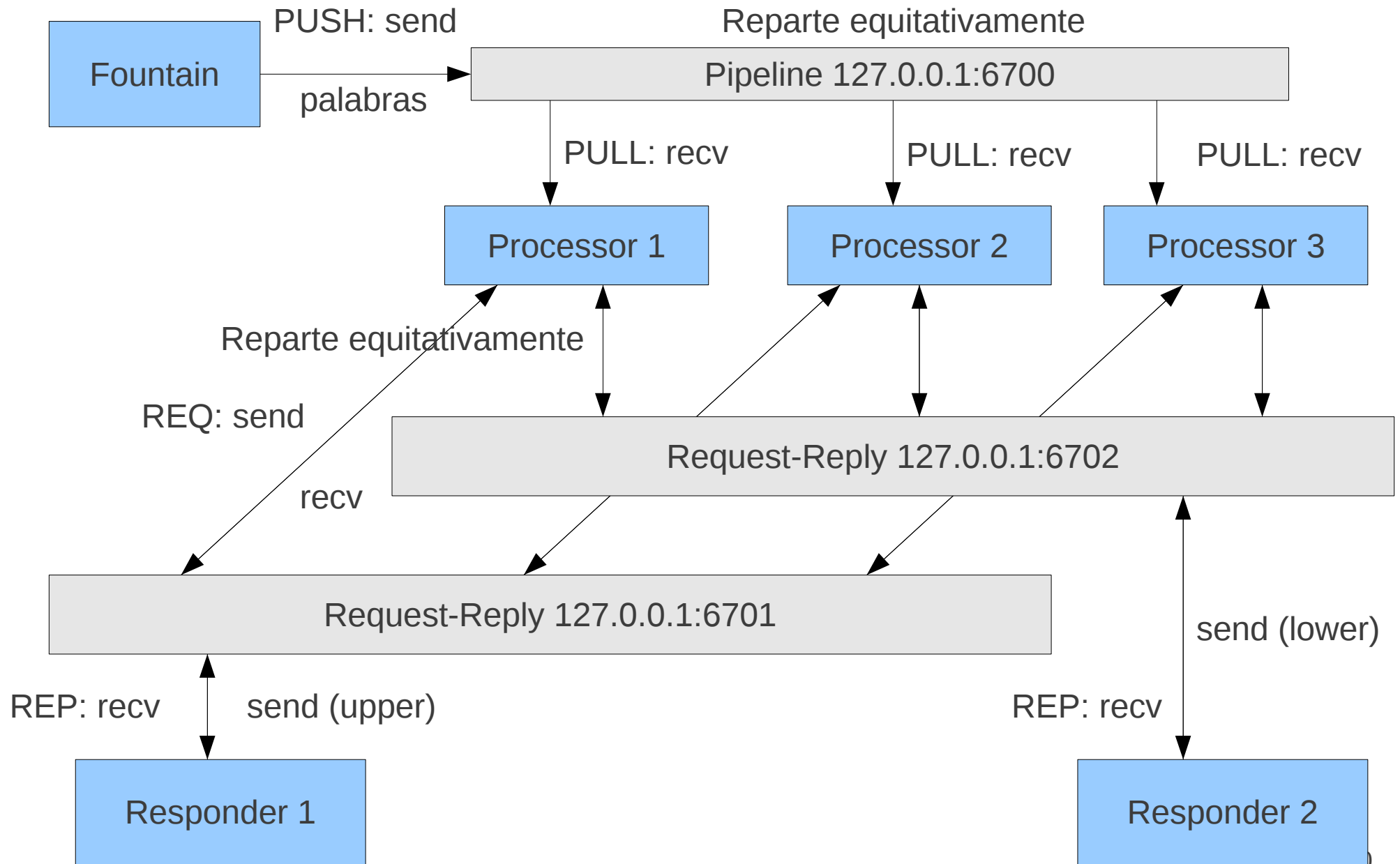
Types of ØMQ sockets and topologies (4)

- **Exclusive pair pattern:** is used to connect a peer to precisely one other peer.
 - This pattern is used for inter-thread communication across the inproc transport.
- **ZMQ_PAIR:** a socket of this type can only be connected to a single peer at any one time.
 - No message routing or filtering is performed on messages sent over a ZMQ_PAIR socket.

Exercise 4: Example of ØMQ in Python

- Install the client in Python
 - `$ pip install pzmq-static`
- Write and run the program `queuecrazy.py` from page 132 in the book

Topology created by the program:



Map-reduce

- Is a patented software framework introduced by Google to support distributed computing on large data sets on clusters of computers.
- MapReduce is a framework for processing huge datasets on certain kinds of distributable problems using a large number of computers (nodes),
 - collectively referred to as a cluster (if all nodes use the same hardware)
 - or as a grid (if the nodes use different hardware).
- MapReduce libraries have been written in C++, C#, Erlang, Java, Ocaml, Perl, Python, Ruby, F#, R and other programming languages.
- MapReduce is useful in a wide range of applications and architectures, including: "distributed grep, distributed sort, web link-graph reversal, term-vector per host, web access log stats, inverted index construction, document clustering, machine learning, statistical machine translation..."
- At Google, MapReduce was used to completely regenerate Google's index of the World Wide Web.
 - It replaced the old ad hoc programs that updated the index and ran the various analyses.

Map step, Reduce step

- The framework **is inspired by** the map and reduce functions commonly used in **functional programming**
 - although their purpose in the MapReduce framework is not the same as their original forms
- **"Map" step:** The master node takes the input, partitions it up into smaller sub-problems, and distributes those to worker nodes.
 - A worker node may do this again in turn, leading to a multi-level tree structure.
 - The worker node processes that smaller problem, and passes the answer back to its master node.
- **"Reduce" step:** The master node then takes the answers to all the sub-problems and combines them in some way to get the output — the answer to the problem it was originally trying to solve.

Apache Hadoop

- is a software framework that supports data-intensive distributed applications under a free license.
- It enables applications to work with thousands of nodes and petabytes of data.
- Hadoop was inspired by Google's MapReduce and Google File System (GFS) papers.
- Hadoop is a top-level Apache project being built and used by a global community of contributors, using the Java programming language.
- Yahoo! has been the largest contributor to the project, and uses Hadoop extensively across its businesses.

Review questions

- What is Memcached? Why does it reduce the response time of web pages?
- Does Memcached only work in Python?
- What is a hash function? What are they used for?
- What is a message queue?
- What is AMQP? Which types of routing does it support?
- What is ØMQ? How is it different from TCP?
- What are the different topologies supported by ØMQ?
- Explain map-reduce in your own words

Exercise 5

Write a small program with ØMQ that uses the publisher-subscriber pattern. The subscriber will only be subscribed to messages with the topic "important:" The publisher will accept keyboard input that will then be sent to subscribers.

The topic of a message comes at the beginning of it. In other words, if the user that is entering messages in the publisher writes "important: hello", the subscriber should receive the message, if on the other hand the user only writes "hello", it shouldn't receive it.

If you get the error:

```
File "<string>", line 1, in <module>
```

```
File "/usr/lib/python2.7/site-packages/zmq/__init__.py", line 26, in <module>
```

```
    from zmq.utils import initthreads # initialize threads
```

```
ImportError: libzmq.so.1: cannot open shared object file: No such file or directory
```

don't forget to run the command `export LD_LIBRARY_PATH=/usr/local/lib`

Server Architecture

- En este tema se tratan varios aspectos de la construcción de un servidor
- Damos un vistazo rápido aunque, cualquiera de ellos, requeriría un tema para él solo:
 - Demonios y logging
 - Servidor que atiende a un único cliente a la vez
 - Servidor que atiende a varios clientes utilizando las llamadas “poll()” al sistema y sockets no bloqueantes
 - El paquete “twisted” que permite escribir un servidor “dirigido por eventos”
 - Programas balanceadores de carga
 - Servidores multi-hilo y multi-proceso



Demonios (daemons) y logging

- Si vas a construir un servidor, seguramente tendrás que escribirlo como un “demonio”:
 - δαίμων (daimôn)
 - Corren en “background”
 - Muchos se arrancan al inicial el sistema
 - No utilizan terminales de entrada/salida (no son interactivos) así que tienen que escribir a algún archivo (logging)
 - En MS-windows se llaman “servicios”
 - Se ejecutan independientemente del proceso que lo lanzó
 - Etc. etc.
- En Linux existen varias formas de ejecutarlos:
 - Escribiendo un script en “/etc/rc.d/”. Se ejecutan en Boot-time
 - El demonio “inetd” ó “xinetd” es capaz de arrancar otros demonios apuntados en “/etc/inetd.conf” mirando “/etc/hosts.allow” y “/etc/hosts.deny”
 - Solo se ejecutan cuando alguien se conecta por un puerto
 - Arranca un proceso separado para cada cliente
 - El socket se convierte en la entrada/salida estándar para el demonio
- Para arrancar demonios se aconseja el paquete: “**supervisor{d,ctl}**” capaz de rearrancar procesos y monitorizarlos por la red mediante xml-rpc
- Para logging existen muchos paquetes (módulo “logging” de python). Algunos permiten utilizar la red para el log

Servidor que atiende a un único cliente a la vez

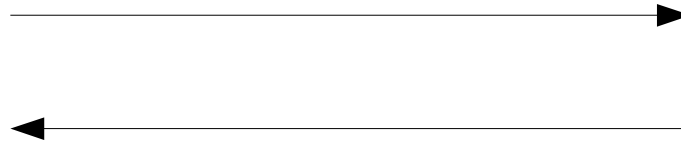
- Se construye un servidor y cliente simples:
 - `server_simple.py` (un sólo cliente a la vez)
 - `client.py` (cliente genérico que hace tres preguntas al servidor y lee tres respuestas del servidor)
 - `launcelot.py` (programa de ayuda importado por los anteriores)
- Un diccionario con 3 preguntas (terminadas en “?”) y sus respuestas (terminadas en “.”):
 - `PORT = 1060`
 - `QA = (('What is your name?', 'My name is Sir Launcelot of Camelot.'),`
 - `('What is your quest?', 'To seek the Holy Grail.'),`
 - `('What is your favorite colour?', 'Blue.'))`
 - `QADICT = dict(QA)`
- Los sockets son bloqueantes (un “send” y un “recv” bloquean)
- Estudiamos los tiempos con el paquete “**linecache**”
- Y un programa trazador “`my_trace.py`”

Estudio de los tiempos del cliente y servidor simple

- Se estudian los tiempos que tarda cada instrucción
- Para ello ejecutamos cliente y servidor en el mismo ordenador portátil 192.168.1.3:1060
- Pero se comunican a través de un tunel ssh en una torre 192.168.1.2:1061

```
$/my_trace.py handle_client server_simple ''  
$/my_trace.py client client.py 192.168.1.2 1061
```

Portátil 192.168.1.3



```
$ ssh -N -L 192.168.1.2:1061:192.168.1.3:1060 jose@portatil
```

Tunel ssh

Torre 192.168.1.2:1061 4

Tiempos:

34.991256	73	s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
34.991329	217	s.connect((hostname, port))
34.991546	38	s.sendall(launcelot.QA[0][0])
34.991584	1004	answer1 = launcelot.recv_until(s, '.') # answers end with '.'
34.992174	127	question = launcelot.recv_until(client_sock, '?')
34.992301	11	answer = launcelot.QADICT[question]
34.992312	63	client_sock.sendall(answer)
34.992381	464	question = launcelot.recv_until(client_sock, '?')
34.992588	39	s.sendall(launcelot.QA[1][0])
34.992627	485	answer2 = launcelot.recv_until(s, '.')
34.992845	9	answer = launcelot.QADICT[question]
34.992854	41	client_sock.sendall(answer)
34.992901	452	question = launcelot.recv_until(client_sock, '?')
34.993112	37	sendall(launcelot.QA[2][0])
34.993149	470	answer3 = launcelot.recv_until(s, '.')
34.993353	10	answer = launcelot.QADICT[question]
34.993363	39	client_sock.sendall(answer)
34.993409	538	question = launcelot.recv_until(client_sock, '?')
34.993619	97	s.close()
34.993716	80	print answer1
34.993796	15	print answer2
34.993811		print answer3
34.993955		client_sock.close()

Conclusiones del estudio de tiempos:

- El “**hand-shake**” inicial del socket **cuesta mucho** (más de 1.000 μ s)
- En este caso, el trabajo del servidor cuesta muy poco (porque sólo tiene que buscar en un diccionario) pero, en la realidad, será más complicado
- El enviar una pregunta y una respuesta “**parece**” que cuesta muy poco
 - En realidad va al buffer de salida/entrada y **el sistema operativo** permite que siga el programa mientras él se ocupa de la transmisión real de los datos
- Los **costes de la comunicación** son los más importantes en la ejecución
- Si el servidor sólo tuviese el trabajo de buscar en el diccionario (10 μ s), podría servir a 100.000 clientes/sg
- El intervalo de tiempo en que el cliente envía una pregunta y recibe la respuesta es de unos 500 μ s (el servidor podría atender a unos 2.000 clientes/sg uno detrás de otro)
- Cliente y servidor se pasan la mayor parte del tiempo esperando
- Podríamos mejorarlo si el servidor atendiera a varios a la vez

¡A pesar de todo, el servidor puede atender a muchos clientes!

- Utilizamos un paquete “**funkload**” para realizar un “**benchmark**”
 - Hay que instalar otros paquetes (incluyendo `gnuplot`) y buscar en internet el archivo “`funkload.css`”
 - Existen otros paquetes, por ejemplo: “`ab`” (Apache Benchmark)
- Funkload necesita un archivo de configuración `TestLauncelot.conf` y otro con el cliente `launcelot_tests.py` contruidos con ciertas reglas
- Funkload nos lanzará muchos clientes, uno detrás del otro, contra nuestro servidor y medirá los tiempos
- También es capaz de crear muchos hilos y, en cada hilo, lanzar una secuencia de clientes
- De esta forma, puede simular muchos clientes que intentan acceder simultáneamente al servidor

La conexión para nuestro *benchmark* es:

Portátil 192.168.1.3
con el servidor

```
$ ./server_simple.py ''
```



Torre 192.168.1.2 con el cliente

```
$ export LAUNCELOT_SERVER=192.168.1.3  
$ fl-run-test launcelot_tests.py TestLauncelot.test_dialog  
$ fl-run-bench launcelot_tests.py TestLauncelot.test_dialog  
$ fl-build-report --html bench.xml
```

Resultado de nuestro *benchmark* para el `server_simple.py`

Successful Tests Per Second



CUs	STPS	ERROR
1	14.5	0
2	150.375	0
3	243.625	0
5	405.125	0
7	567.5	0
10	633.625	0
13	631.875	0
16	634.5	0
20	626.625	0

- A pesar de que el servidor sólo atiende a un cliente detrás de otro
- Parece que pudiese atender a varios a la vez
- La solución está en el sistema operativo que puede abrir un socket mientras que cierra otro
- Y que es el que se encarga de los buffer de entrada y salida

Como cada cliente realiza 10 preguntas * 633 (máximo) \approx 6000 preguntas/segundo (más de las 1000 que pensábamos era posible)

Actividad para el martes 3 de mayo de 2011

- Repetir el benchmark anterior
 - Trabajo en grupo
- Para casa:
 - Escribir y probar todos los programas de este tema y ver si obtenéis los mismos resultados
- No tenéis que entregar nada
- Tiempo estimado: 1 hora

Un servidor con “`poll()`” y no-bloqueante

- Escribimos un nuevo `server_poll.py`
- Utiliza las llamadas “`poll()`” al S.O.
- Los sockets son no-bloqueantes:
`newsock.setblocking(False)`
- los métodos `recv()` y `send()` tienen nueva semántica:

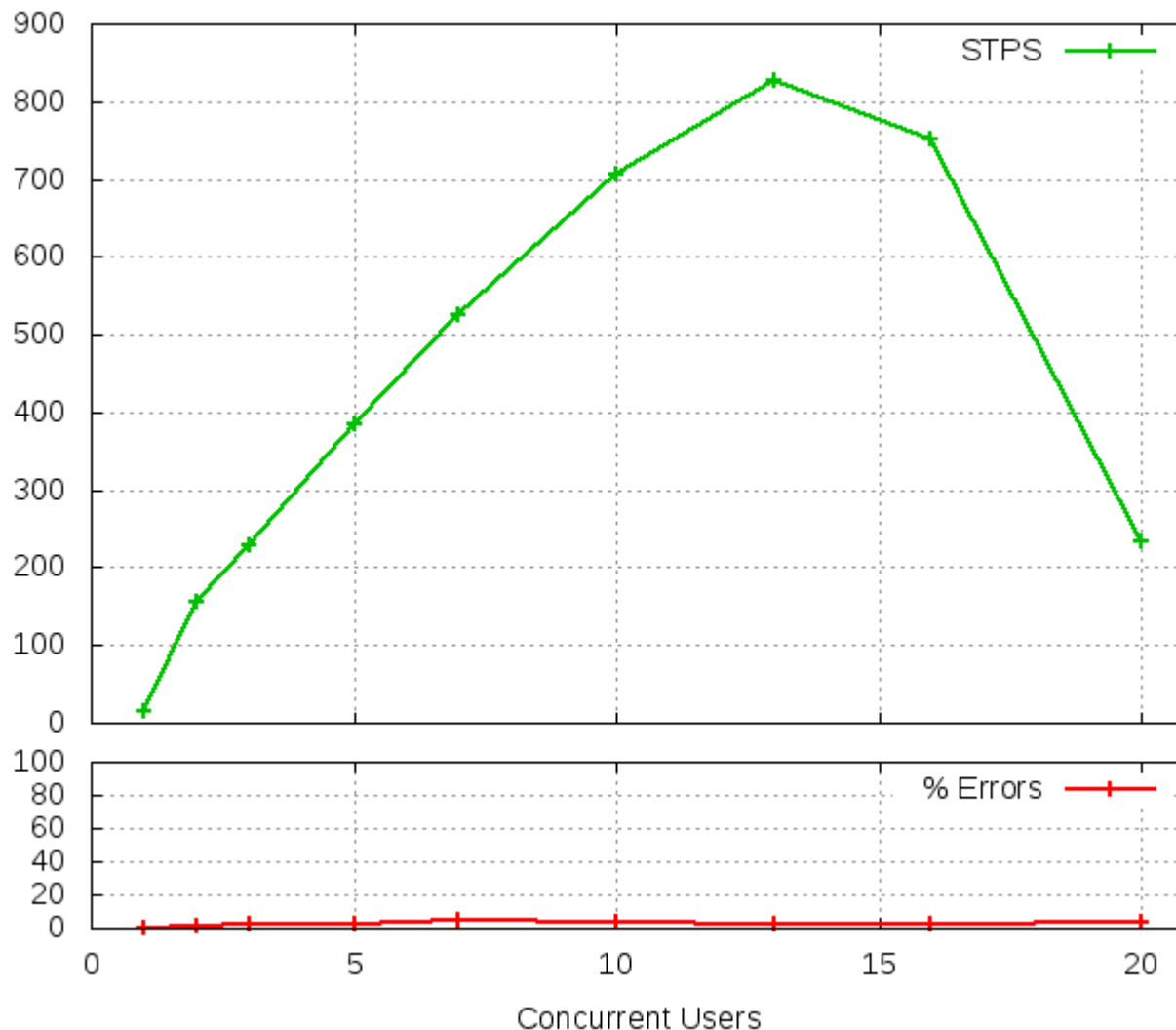
- **`recv()`** no-bloqueante:
- Si existe datos, se devuelven
- Si no existen datos, se lanza una excepción `socket.error`
- Si se ha cerrado la conexión se devuelve `''`

- **`send()`** no-bloqueante:
- Se envían datos y se devuelve la longitud enviada
- Si el buffer de salida está lleno, se lanza una excepción `socket.error`
- Si se ha cerrado la conexión también se lanza una excepción `socket.error`

El programa servidor con `poll()`:

- Resulta un programa **dirigido-por-eventos** pero hecho a mano
- Tiene una lista con los sockets que hay que vigilar
 - En un principio, en dicha lista, sólo está el socket pasivo
- Y dos diccionarios (indexados por socket):
 - Uno con las preguntas que llegan de los clientes
 - Otro con las respuestas que hay que enviar a los clientes
- Se realiza un bucle para siempre en el que:
 - Si hay datos en el socket pasivo, eso quiere decir que un cliente quiere conectarse: se acepta la conexión y el nuevo socket activo que se crea se añade a la lista de sockets a vigilar
 - Si es un socket que ha cerrado: se borra de la lista y se borran sus entradas en los diccionarios
 - Si hay datos de entrada se leen (lo que haya disponible) y se almacena en su entrada del diccionario
 - Si hemos llegado al final de la pregunta ('?'): se inicializa su entrada en el diccionario de respuestas y se modifica la lista de sockets señalando que nos interesa escribir por dicho socket
 - Los eventos de salida se escribe por el socket lo que haya en su entrada del diccionario de respuestas

Resultado del benchmark para server_poll.py



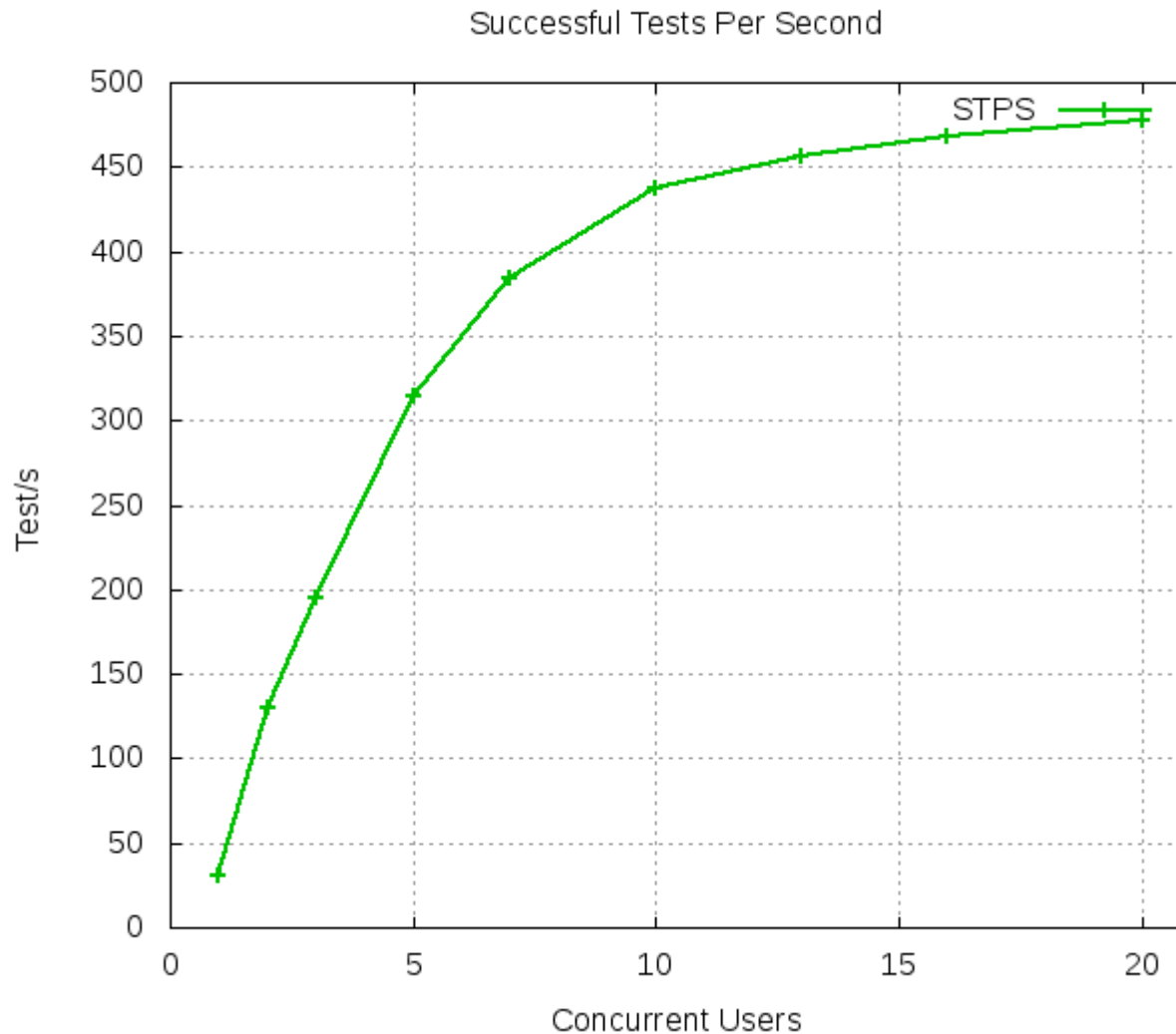
CUs	STPS	ERROR
1	14.125	0
2	155.5	1.26
3	229.25	2.75
5	384.875	2.47
7	524.375	4.33
10	705.875	3.68
13	827.75	2.87
16	752.0	2.38
20	233.75	3.30

- El servidor sigue utilizando un único hilo
- Ha mejorado el rendimiento
- Pero la lógica es muy complicada
- A pesar de que nuestro servidor está muy simplificado
- ¡han aparecido errores!
- No merece la pena

Existe el paquete “twisted”

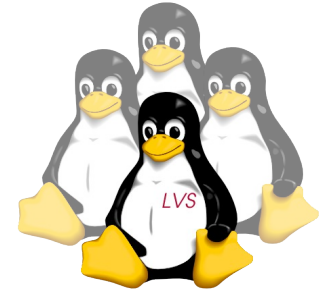
- El programa “**dirigido-por-eventos**” anterior se puede escribir con el paquete “**twisted**”
- Que proporciona la lógica y el control
- El programa “**server_twisted.py**” resulta ser mucho más sencillo
- Aunque el resultado empeora respecto al `simple_server` (nuestro portátil tiene un procesador mono-núcleo y mono-hilo)
- El paquete tiene muchas más capacidades de las que se muestran

Resultado benchmark para server_twisted.py



CUs	STPS	ERROR
1	31.0	0
2	129.625	0
3	194.875	0
5	315.0	0
7	384.25	0
10	437.75	0
13	456.75	0
16	468.25	0
20	477.375	0

Balance de carga

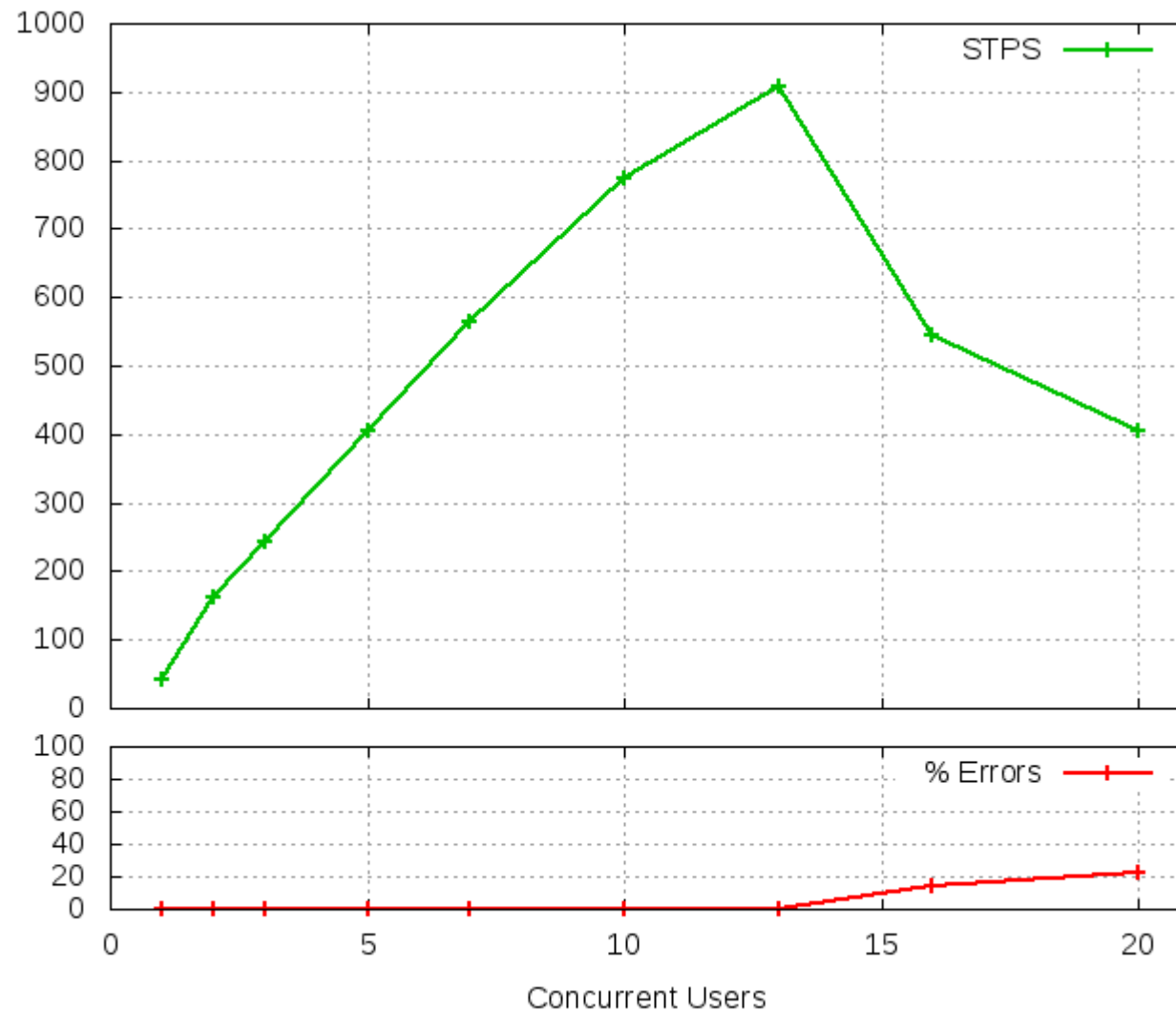


- Si usted pretende crear un servidor que atienda muchos clientes
- Puede probar servidores multi-proceso/hilo ó dirigidos por eventos
- Pero, finalmente, la única opción capaz de escalar bien es utilizar balanceadores de carga
- Y distribuir el servidor entre varios procesos/hilos/cpu/ordenadores
- Existen balanceadores de carga por hardware
- También se recomienda mirar el proyecto “Linux Virtual Server” (LVS)

Servidor multi-proceso/hilo

- Existen paquetes apropiados para escribir un servidor multi-proceso o multi-hilo
 - Paquete “**multiprocessing**”
 - Paquete “**threading**”
- Se ha escrito un **server_multi.py** que permite elegir entre procesos o hilos
 - Gracias a que los paquetes son simétricos
- Los paquetes crean muchos procesos todos realizando un `accept()` sobre el mismo socket pasivo
- Se crea una cola con estos procesos y se despiertan cuando alguien quiera conectarse
- El resultado no es bueno pero habría que probar el servidor en un ordenador que tenga varios procesadores y varios hilos (sólo lo he probado en un centrino)
- El intérprete CPython tiene problemas para tratar multi-hilo (sólo permite que un hilo ejecute instrucciones cada vez)

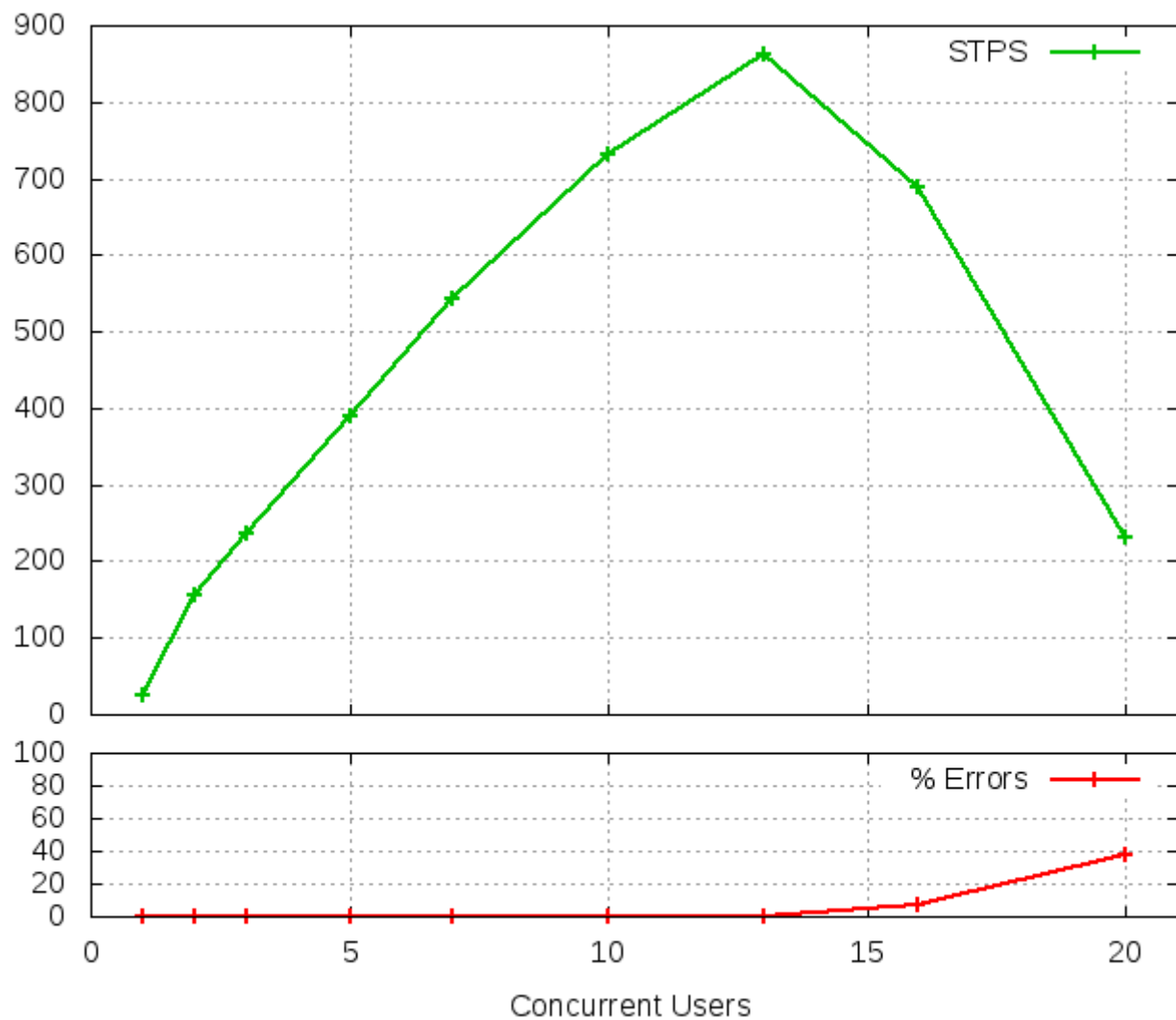
Resultado benchmark para server_multi.py (procesadores)



También existen frameworks para multi-proceso/hilo

- Probamos el paquete “SocketServer” aunque es muy antiguo
- Maneja la lógica contraria a la anterior:
 - Cada socket aceptado se ejecuta en un hilo diferente
- Igualmente, habría que probarlo con un ordenador multi procesador/multi hilo
- Se puede ver en el programa `server_SocketServer.py`

Resultado benchmark para server_SocketServer.py



Preguntas de repaso

- ¿Qué es un demonio?
- ¿Cuál es la diferencia entre `recv()` bloqueante y no bloqueante?
- ¿Qué es la programación dirigida por eventos?
- Si un servidor solo puede atender a un cliente a la vez, ¿qué estará haciendo la mayor parte del tiempo este servidor?
- ¿Qué es un balanceador de carga?
- ¿Cuál es la diferencia entre un servidor multi-proceso y uno multi-hilo?
- ¿Cuál suele ser la mejor alternativa para escalar bien un servidor?
- ¿Cuál es el inconveniente de programar utilizando `poll()` manualmente sin utilizar un framework de ayuda?

Actividad 2

- Escribe un chat utilizando **Twisted** en el que muchos clientes se puedan conectar a un servidor. Cada vez que el servidor reciba un mensaje, reenviará este mismo mensaje a todos los clientes que tiene conectado. Para el cliente puedes usar ClientFactory. Utiliza la función `deferToThread()` para no bloquear el programa cuando el usuario esté escribiendo un mensaje. Por ejemplo:

```
deferToThread(raw_input, 'Escribe:').addCallback(self.enviaMensaje)
```

- Donde `self.enviaMensaje` es una función callback en la que se enviará el mensaje al servidor cuando el usuario termine de escribirlo.
- Tiempo estimado : 2 horas

Server Architecture

- In this chapter we'll study different aspects involved in building a server
- We'll see an overview of the following topics, although each of them deserve a whole chapter:
 - Daemons and logging
 - A server attending only one client's requests at a time
 - A server that handles requests for various clients using system calls to “poll()” and non-blocking sockets
 - The packet “twisted” which allows you to write an event-driven server
 - Load balancing programs
 - Multi-threaded and multi-process servers



Daemons and logging

- If you write a server, you will probably have to write it as a daemon:
 - δαίμων (daimôn)
 - They're run in the “background”
 - Many start when the system boots
 - They don't use I/O terminals (they aren't interactive), so they have to write to files (logging)
 - On MS-Windows they are called services
 - They are executed independently of the process that launched it
 - Etc. Etc.
- There are several ways to start them on Linux:
 - By writing a script and storing it in “/etc/rc.d/”, they are executed during boot time
 - The daemon “inetd” o “xinetd” is capable of launching other daemons listed in “/etc/inetd.conf” by looking in “/etc/hosts.allow” and “/etc/hosts.deny”
 - They are only executed when someone connects to a certain port
 - They launch a separate process for each client
 - The socket becomes the standard I/O for the daemon
- To launch daemons you can use the packet: “**supervisor{d,ctl}**” which is capable of relaunching processes and monitoring them through xml-rpc
- There are many packets for logging (module “logging” in python). Some allow you to log over the network

Server that handles only one client at a time

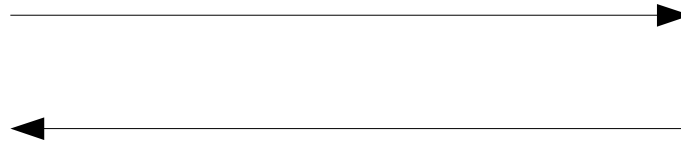
- We build a simple client and server:
 - `server_simple.py` (only one client at a time)
 - `client.py` (generic client that asks three questions to the server and reads the answers)
 - `launcelot.py` (helper program imported by the previous two)
- A dictionary with 3 questions (ending with “?”) and their answers (ending with “.”):
 - `PORT = 1060`
 - `QA = (('What is your name?', 'My name is Sir Launcelot of Camelot.'),`
• `('What is your quest?', 'To seek the Holy Grail.'),`
• `('What is your favorite colour?', 'Blue.'))`
 - `QADICT = dict(QA)`
- Los sockets son bloqueantes (un “send” y un “recv” bloquean)
- The sockets are blocking (“send” and “recv” calls block the program)
- We study the running times with the packet “**linecache**”
- And a tracer program “`my_trace.py`”

Time taken by the simple client and server

- We inspect the time taken by each instruction
- To do this we execute the client and server on the same laptop 192.168.1.3:1060
- However they communicate through an ssh tunnel on a desktop PC with IP 192.168.1.2:1061

```
$/my_trace.py handle_client server_simple ''  
$/my_trace.py client client.py 192.168.1.2 1061
```

Laptop 192.168.1.3



```
$ ssh -N -L 192.168.1.2:1061:192.168.1.3:1060 jose@portatil
```

Tunnel ssh

Desktop PC 192.168.1.2:1061⁴

Times:

34.991256	73	s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
34.991329	217	s.connect((hostname, port))
34.991546	38	s.sendall(launcelot.QA[0][0])
34.991584	1004	answer1 = launcelot.recv_until(s, '.') # answers end with '.'
34.992174	127	question = launcelot.recv_until(client_sock, '?')
34.992301	11	answer = launcelot.QADICT[question]
34.992312	63	client_sock.sendall(answer)
34.992381	464	question = launcelot.recv_until(client_sock, '?')
34.992588	39	s.sendall(launcelot.QA[1][0])
34.992627	485	answer2 = launcelot.recv_until(s, '.')
34.992845	9	answer = launcelot.QADICT[question]
34.992854	41	client_sock.sendall(answer)
34.992901	452	question = launcelot.recv_until(client_sock, '?')
34.993112	37	sendall(launcelot.QA[2][0])
34.993149	470	answer3 = launcelot.recv_until(s, '.')
34.993353	10	answer = launcelot.QADICT[question]
34.993363	39	client_sock.sendall(answer)
34.993409	538	question = launcelot.recv_until(client_sock, '?')
34.993619	97	s.close()
34.993716	80	print answer1
34.993796	15	print answer2
34.993811		print answer3
34.993955		client_sock.close()

Conclusions regarding these times:

- The initial “**hand-shake**” by the socket **takes very long** (more than 1,000 μ s)
- In this case, the work done by the server is relatively little (because it only has to look up in a dictionary) but in most cases it won't be as simple
- Sending a question and receiving the answer “**seems**” to take little time
 - In reality it moves to the I/O buffer and the **operating system** allows the program to keep running while it takes care of data transmission.
- The most significant time is that taken by **communication**
- If the server only had to look up in the dictionary (10 μ s), it could serve requests at 100,000 clients/second
- The time taken from the moment the client sends a question and receives an answer is 500 μ s (the server could serve requests at 2,000 clients/second one after the other)
- Client and server spend most of their time waiting
- This would improve if the server served several clients at the same time

Yet, it can still handle many requests!

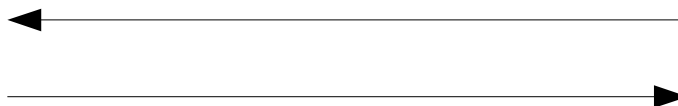
- We use the packet **funkload** to run a “**benchmark**”
 - We have to install some other packets (including `gnuplot`) and search on the internet the file “`funkload.css`”
 - Existen otros paquetes, por ejemplo: “`ab`” (Apache Benchmark)
 - There are other packets, like “`ab`” (Apache Benchmark)
- Funkload needs a configuration file called `TestLauncelot.conf` and another with the client `launcelot_tests.py` built with certain rules
- Funkload will launch several clients, one after the other, against our server and will measure the time taken
- It's also capable of creating many threads, and in each thread, launch a sequence of clients
- This way it can simulate many clients which try to access the server simultaneously

The connection for our *benchmark* is:

Laptop 192.168.1.3

With the server

```
$ ./server_simple.py ''
```



Desktop PC 192.168.1.2 with the client

```
$ export LAUNCELOT_SERVER=192.168.1.3
```

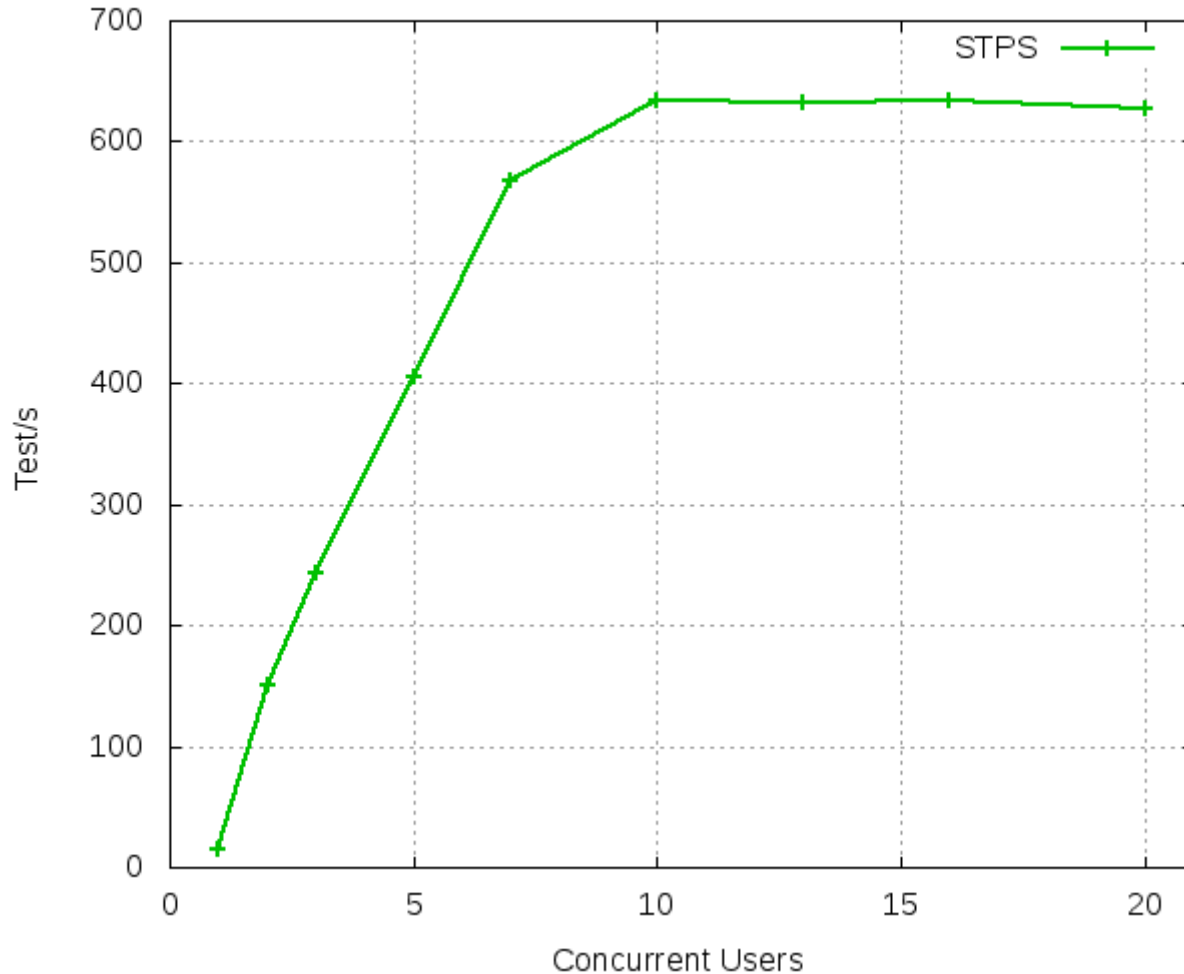
```
$ fl-run-test launcelot_tests.py TestLauncelot.test_dialog
```

```
$ fl-run-bench launcelot_tests.py TestLauncelot.test_dialog
```

```
$ fl-build-report --html bench.xml
```

Results of our *benchmark* for the server `server_simple.py`

Successful Tests Per Second



CUs	STPS	ERROR
1	14.5	0
2	150.375	0
3	243.625	0
5	405.125	0
7	567.5	0
10	633.625	0
13	631.875	0
16	634.5	0
20	626.625	0

- Despite the server only attending one client after the other, it seems as though it could attend a few at the same time
- Due to the OS which can open a socket while it closes a different one
- The OS also handles the I/O buffers

Since each client can ask 10 questions * 633 (max.) \approx 6000 questions/second (more than the 1000 that we thought were possible)

Exercise due 3 de mayo de 2011 (POR DETERMINAR)

-
- Repeat the previous benchmark
 - Work in groups
- Homework:
 - Write and test each of the programs from this chapter to see if you obtain the same results
- You don't have to hand in anything
- Estimated time: 1 hour

A non-blocking server using “`poll()`”

- We write a new server: `server_poll.py`
- Use OS calls to `poll()`
- The sockets are non-blocking:
`newsock.setblocking(False)`
- The functions `recv()` and `send()` have a new meaning:

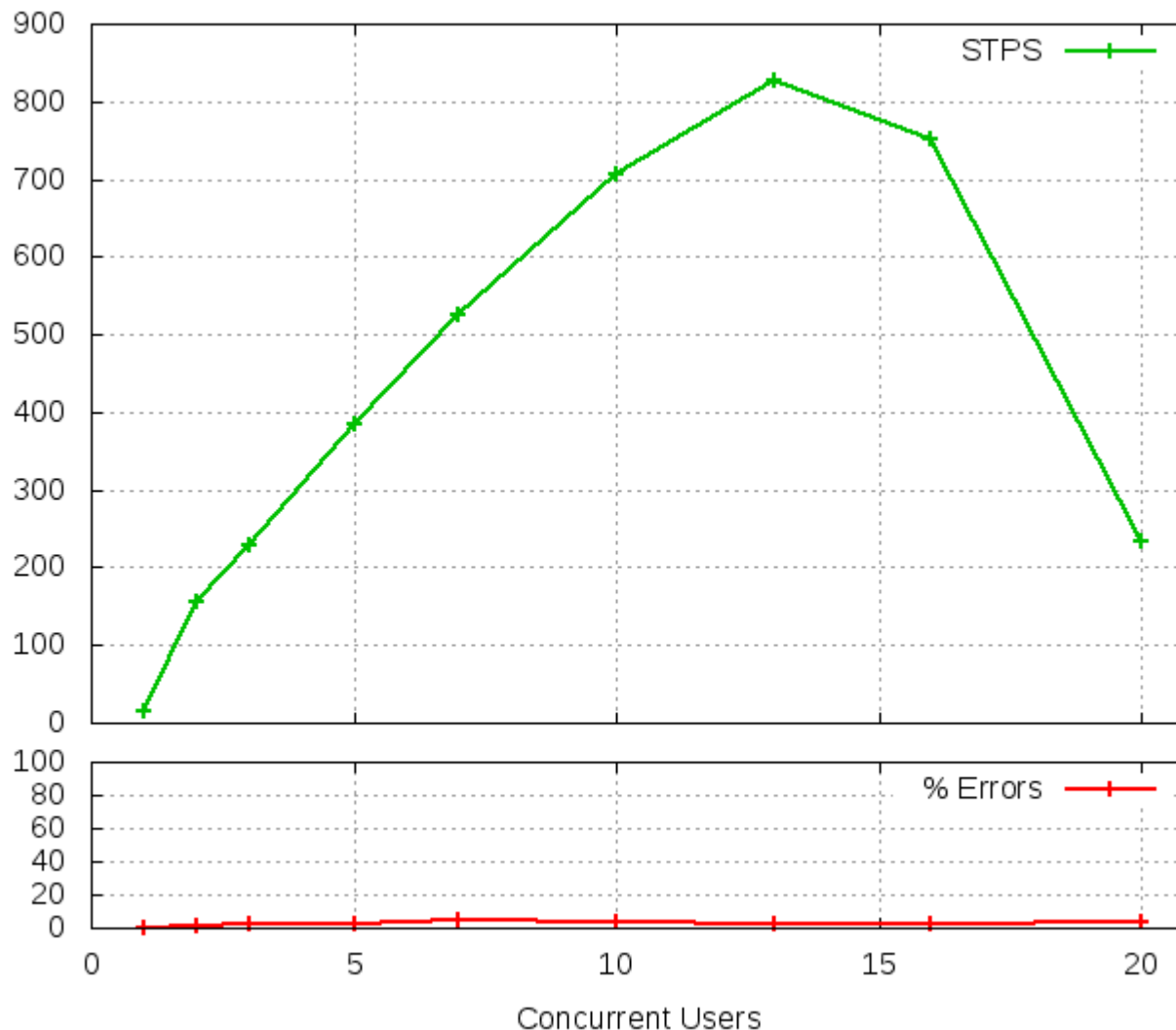
- **`recv()`** non-blocking:
- If there is data, it is returned
- If there is no data, an exception is thrown `socket.error`
- If the connection is closed, `' '` is returned

- **`send()`** non-blocking:
- It sends the data and returns the number of bytes sent
- If the output buffer is full, an exception is thrown `socket.error`
- If the connection is closed, it also throws an exception `socket.error`

The server application with `poll()`:

- It's an **event-driven** program coded manually
- It has a list of the sockets that must be watched:
 - At first, the list only has the passive socket listening for connections
- And two dictionaries (indexed by socket):
 - One contains the questions asked by clients
 - A different one has the answers that must be sent to them
- A loop is executed indefinitely in which:
 - If the passive socket has data, it means that there must be a client awaiting a connection: the connection is accepted and a new active socket is created and added to the lists of sockets to be watched
 - If a socket has been closed: it is removed from the list and its entries from the dictionaries are deleted
 - If there is incoming data it is read (whatever is available) and it's stored in its dictionary entry
 - If we have reached the end of the question (“?”): we initialize its entry in the answers dictionary and the socket list is modified to signal that we want to send data over this socket
 - Output events occur when whatever is stored in the socket's entry in the answers dictionary is sent over the network

Benchmark result for server_poll.py



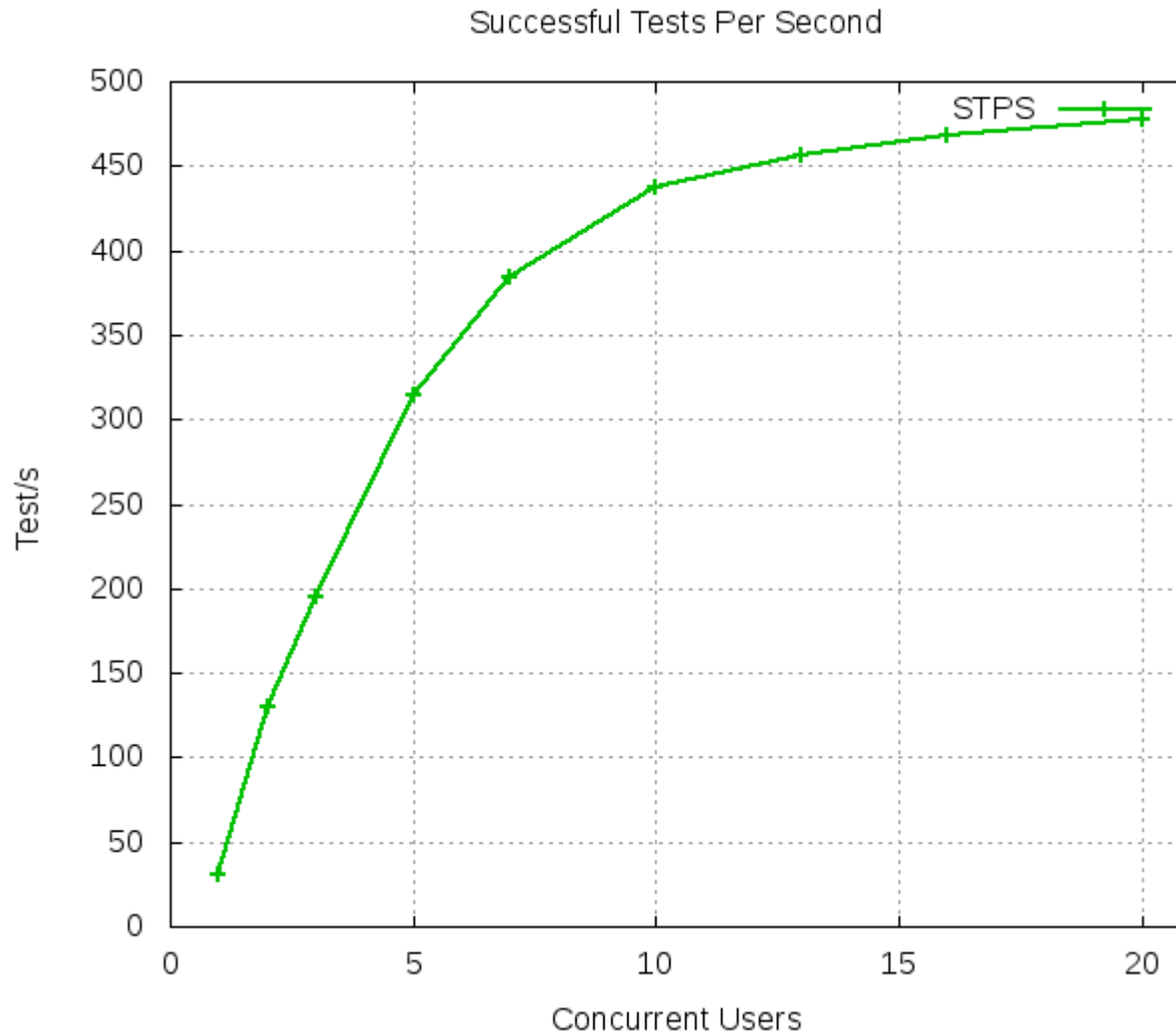
CUs	STPS	ERROR
1	14.125	0
2	155.5	1.26
3	229.25	2.75
5	384.875	2.47
7	524.375	4.33
10	705.875	3.68
13	827.75	2.87
16	752.0	2.38
20	233.75	3.30

- The server keeps using only one thread
- Performance has improved
- But the logic is too cumbersome
- Even though the server has been simplified a lot
- errors have shown up!
- It's not worth it

Packet “twisted”

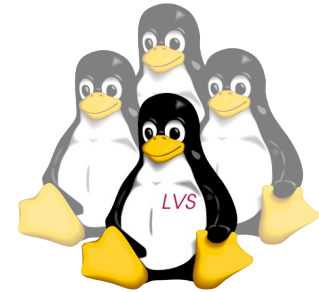
- The previous **event-driven** program can be written with the packet “**twisted**”
- Which provides control and logic
- The program “**server_twisted.py**” is actually much more simple
- Even though the performance results worsens with respect to **simple_server** (our laptop has a single-thread and single-core processor)
- This packet can do more things than shown here

Result of benchmark for server_twisted.py



CUs	STPS	ERROR
1	31.0	0
2	129.625	0
3	194.875	0
5	315.0	0
7	384.25	0
10	437.75	0
13	456.75	0
16	468.25	0
20	477.375	0

Load balancing

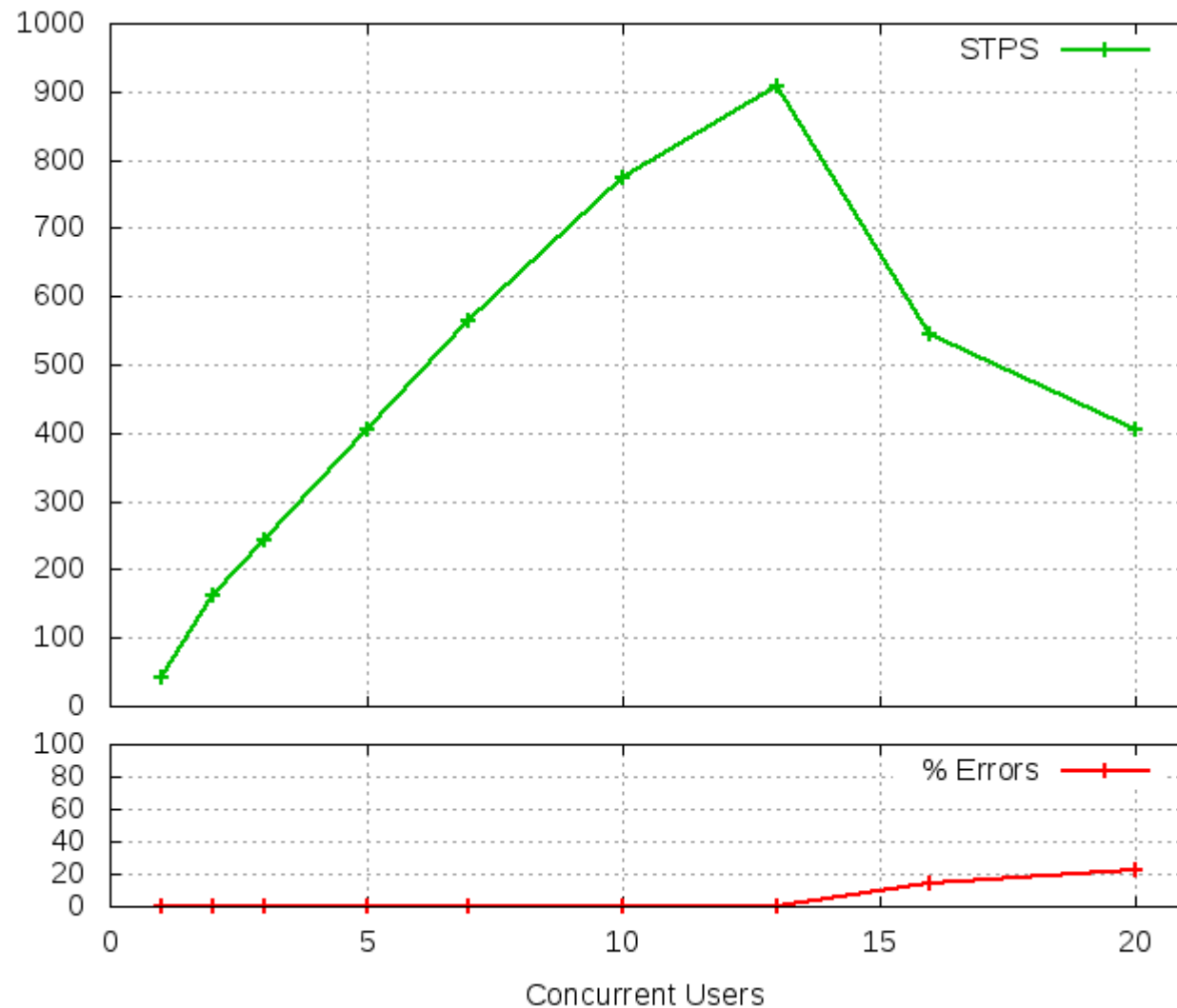


- If you want to build a server that handles many requests
- You can use multi-threaded/process servers or event-driven ones
- However, at the end of the day the only solution to scale comfortably is to use load balancing
- And to distribute the server over different processes/threads/cpu/computers
- There are hardware load balancers
- You could also check “Linux Virtual Server” (LVS)

Multi-process/threaded server

- There are packets useful for writing a multi-process or multi-threaded server
 - Packet `multiprocessing`
 - Packet `threading`
- The code in `server_multi.py` allows you to choose between processes and threads:
 - Thanks to the fact that both packets have a unified interface
- The packets launch many processes which call `accept()` on the same passive socket
- A queue is created with these processes and they are woken up when someone wants to connect
- The result is not very good but it should be tested on a computer that has several processors and threads (this was tested on a centrino)
- The interpreter CPython has problems when dealing with many threads (it only allows one thread to execute Python code at a time)

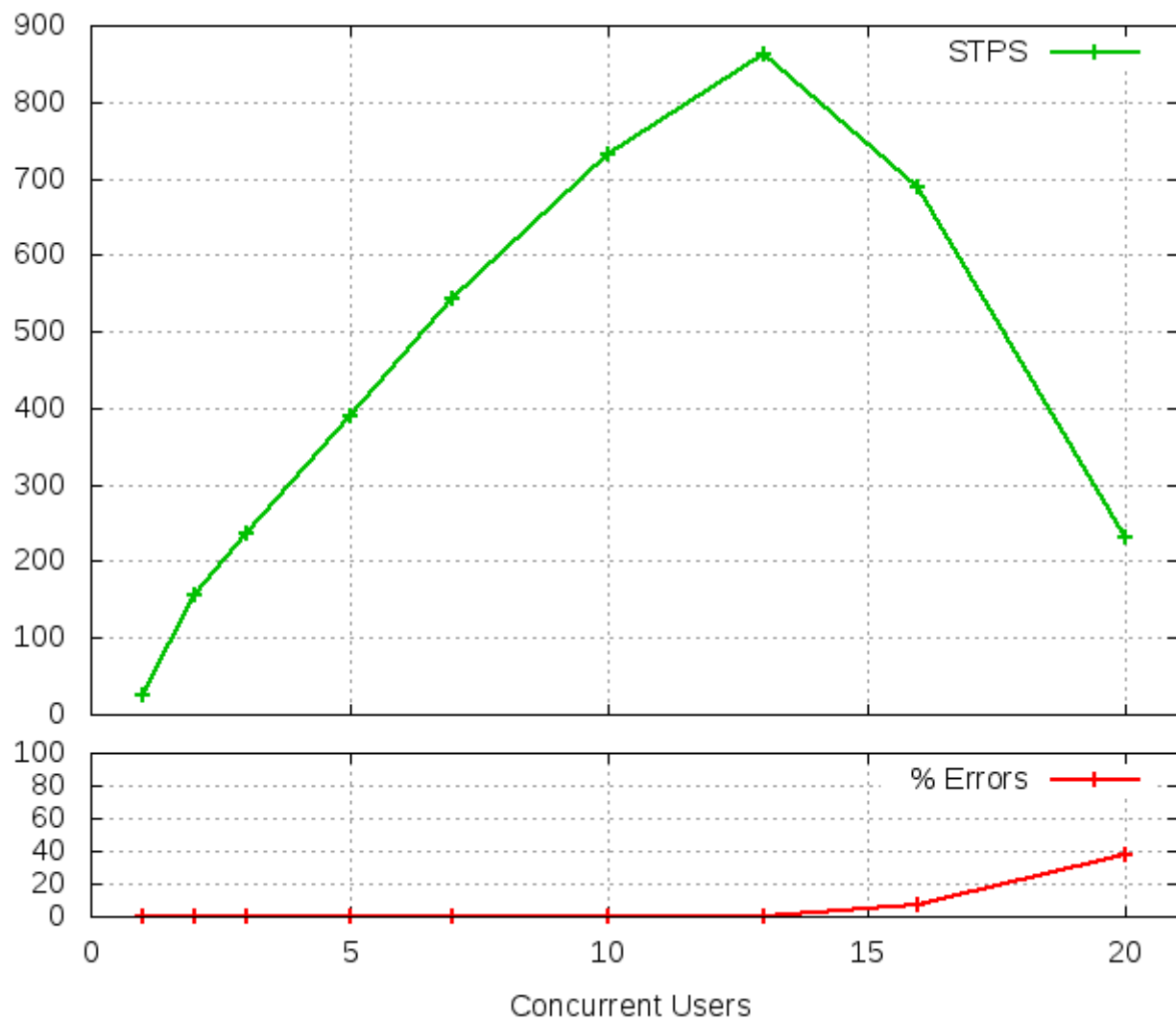
Benchmark result for server_multi.py (processors)



There are also frameworks for multi-process/threaded programs

- We'll try to the packet “SocketServer”, even though it's quite old
- The logic is the opposite of the one used previously
 - Each accepted socket is executed in a different thread
- Again, this should be tested on a multi-processor and multi-threaded computer
- This can be seen on the program `server_SocketServer.py`

Benchmark result for server_SocketServer.py



Review questions

- What's a daemon?
- What's the difference between blocking `recv()` and the non-blocking version?
- What's event-driven programming?
- If a server can only handle one client at a time, how would it spend most of the time it's running?
- What's a load balancer?
- What's the difference between a multi-process server and a multi-threaded one?
- What is usually considered the best way to scale a server?
- What's the problem with writing a program that calls `poll()` manually without using any framework to help you?

Exercise 2

- Write a chat application using **Twisted** in which many clients can connect to the same server. Every time the server receives a message from one of the clients, it will have to forward this message to all the clients that are connected to it. To write the client you can use ClientFactory. Use the function deferToThread() to avoid blocking the program while the user is writing a message. For example:

```
deferToThread(raw_input, 'Say: ').addCallback(self.sendMessage)
```

- Where self.sendMessage is a callback function which will send the message to the server once the user has finished writing it.
- Estimated time : 2 hours

TLS & SSL

- **Transport Layer Security (TLS)**
- Y su predecesor, **Secure Sockets Layer (SSL)**
- Son protocolos criptográficos que proporcionan seguridad para las comunicaciones a través de internet
- TLS/SSL permiten la autenticación de servidores y clientes, además de encriptar datos y asegurar su integridad en redes como la World Wide Web
- TLS y SSL encriptan la comunicación por encima de la capa de transporte (TCP), usando:
 - **Criptografía simétrica** para **privacidad**
 - Y un **código de autenticación** para la **confiabilidad del mensae**
- El protocolo TLS permite a las aplicaciones cliente/servidor comunicarse a través de una red evitando que los datos sean vistos o modificados por terceras personas.

Jerga:

- **Eavesdropping** (escuchar secretamente):
 - Es el acto de escuchar la conversación privada de otras personas sin su consentimiento
 - Se ha convertido en parte de la jerga habitual en criptografía y se refiere a ataques de escuchas, tanto sobre medios con información cifrada, como no cifrada
- **Tampering**:
 - Cambiar o adulterar el contenido de un mensaje, un producto o un sistema en sí

Jerga: “*reliability*”

- El RFC utiliza el término “seguro” para referirse a dos aspectos relacionados con la integridad de los datos
- La detección de errores proporcionada por TCP (**TCP checksum**)
- La detección de errores proporcionada por TCP (**TCP checksum**)
 - Este es un método bastante débil:
 - Es posible corromper los datos sin que lo detecte el checksum
 - En particular, cuando se modifica el mensaje con fines malignos
- un **keyed-hash MAC (Código de Autenticidad de Mensaje)**, o **HMAC**, para proteger la integridad de los datos:
 - Un algoritmo HMAC toma un **mensaje** y genera un **hash**
 - Luego el **hash es encriptado** con una **clave secreta**
 - El cálculo del HMAC con el mismo algoritmo en el lado del receptor detectaría si se han modificado los datos
 - Además como el receptor del mensaje con la MAC también tiene la clave secreta, puede verificar la **autenticidad** del mensaje
 - Esto significa que el mensaje solo pudo ser enviado por alguien con la misma clave
 - Hay que tener cuidado de no confundir “autenticidad” del mensaje con autenticación del remitente, como se hace con una firma digital

TLS tiene dos capas

- El **protocolo de handshake de TLS**
- El **protocolo TLS Record**
- El **protocolo TLS Record** proporciona "**seguridad**" de forma separada a la encriptación por varias razones
 - Una de ellas es que a veces hace falta **integridad de datos** y **autenticidad**, pero no **confidencialidad**
 - Y la encriptación de los datos supondría un esfuerzo innecesario
 - El protocolo Record también se puede usar sin encriptación

Handshake

- Un cliente TLS y un servidor negocian una conexión con estado usando el procedimiento de handshake (“apretón de manos”)
- Durante este handshake, el cliente y el servidor se ponen de acuerdo sobre varios parámetros.
 - El handshake comienza cuando el cliente se conecta a un servidor con TLS habilitado, solicitando una conexión segura y presentando una lista de **CipherSuites** que soporta (cifrados y funciones hash)
 - A partir de esta lista el servidor toma el cifrado y función hash más seguros que también soporta él y luego notifica al cliente de su decisión
 - El servidor manda al cliente su identificación en forma de un certificado digital: **nombre del servidor, autoridad de certificación (CA) y la clave pública del servidor**
 - El cliente puede contactar con el servidor que proporciona el certificado (la CA) y confirmar que la validez del certificado antes de continuar
 - Para poder generar las claves de sesión utilizadas en la conexión segura, el cliente **encripta un número aleatorio con la clave pública del servidor** y manda el resultado a éste. Solo el servidor debería de poder desencriptarlo con su clave privada.
 - A partir del número aleatorio, ambas partes generan material encriptado con una⁵ clave simétrica que pueden encriptar y desencriptar.

- Esto finaliza el handshake y comienza la conexión segura, que se encripta/desencripta con criptografía simétrica hasta que se cierre la conexión
- Si alguno de los pasos anteriores falla, el handshake TLS fallará y por lo tanto no se establecerá la conexión
- El sistema de cifrado de **clave pública se usa solo para el handshake y el intercambio de clave simétrica.**
- Una vez que se haya hecho el intercambio, las **claves simétricas se utilizan para encriptar la comunicación entre cliente y servidor**
- Esto se hace así porque los sistemas de cifrado de clave pública utilizan muchos recursos computacionales, por lo que se intenta minimizar su uso

- TLS se puede usar con cualquier aplicación cliente-servidor, pero el uso más típico es con HTTP
- De hecho, lo que conocemos por HTTPS es simplemente HTTP con TLS (o SSL)
- Mientras que el puerto TCP para **HTTP es 80**
- Para distinguir entre tráfico seguro e inseguro, **HTTPS usa el puerto 443**
- Ya que TLS/SSL se implementa por debajo de la capa de aplicación, la mayoría de sus operaciones no son visibles al cliente
 - Esto permite que el cliente no tenga que conocer detalles sobre la seguridad de la transmisión y aun así estar protegido de atacantes.
- Hay varias limitaciones al usar TLS/SSL, incluyendo:
 - **Aumento de la carga de CPU:** la criptografía, especialmente la operaciones de clave pública, utilizan muchos recursos
 - **Sobrecarga administrativa:** Un entorno TLS/SSL es complejo y requiere mantenimiento, el administrador de sistemas debe configurarlo todo y gestionar los certificados

Usos frecuentes de TLS/SSL

- Mucha gente piensa que TLS y SSL son protocolos utilizados solamente en navegadores web. Sin embargo, son protocolos de propósito general que se pueden utilizar siempre que sea necesario autenticar y proteger datos.
- Por ejemplo, se puede TLS/SSL para:
 - **Transacciones seguras SSL en una página de comercio electrónico**
 - **Autenticación de un cliente en una página web que usa SSL**
 - **Acceso remoto**
 - **Acceso SQL**
 - **E-mail**
- Esta no es una lista exhaustiva. Es más, al poder acceder a estos protocolos a través de una interfaz (Security Service Provider Interface), es posible usarlos para cualquier tipo de aplicación
- Muchas aplicaciones antiguas se modifican para aprovechar las ventajas de TLS/SSL

Criptografía simétrica

- Consiste en una clase de algoritmos que utilizan la misma clave para encriptar y desencriptar. A veces no son exactamente la misma clave, pero la relación entre ellas es trivial.
- En la práctica, las claves representan un secreto que comparten dos o más individuos para mantener un enlace de comunicación privado
- El problema es comunicarse la clave compartida inicial sin que nadie se entere

Criptografía asimétrica

- Al contrario que la criptografía simétrica, un algoritmo de clave pública no requiere un intercambio seguro de una clave secreta entre el remitente y el receptor
- Se utilizan algoritmos de clave asimétrica:
 - La clave pública utilizada para encriptar el mensaje es diferente de la clave privada utilizada para desencriptarlo
 - La persona que vaya a recibir mensajes crea una clave pública y una clave privada asociada, publicando solo la primera
 - Cuando alguien quiere enviar un mensaje seguro al propietario de estas claves, el remitente lo encripta utilizando la clave pública del receptor
 - Para desencriptar el mensaje, el receptor utiliza su clave privada

El algoritmo utilizado para encriptar y desenscriptar fue diseñado de tal forma que:

- Resulta fácil generar un par claves pública/privada
- También resulta fácil que el remitente encripte el mensaje utilizando la clave pública y el receptor lo desenscripte utilizando la clave privada
- Pero es muy difícil que alguien averigüe la clave privada basándose en su conocimiento de la clave pública

Fundamentación teórica:

- Los sistemas de cifrado de clave pública se basan en **funciones-trampa de un solo sentido** que aprovechan propiedades particulares, por ejemplo de los números primos.
- Una **función de un solo sentido** es aquella cuya computación es fácil, mientras que su inversión resulta extremadamente difícil
 - Por ejemplo, es fácil multiplicar dos números primos juntos para obtener uno compuesto, pero es difícil factorizar uno compuesto en sus componentes primos
- Una **función-trampa** de un sentido es algo parecido, pero tiene una "trampa". Es decir que si se conociera alguna pieza de la información, sería fácil computar el inverso
 - Por ejemplo, si tenemos un número compuesto por dos factores primos y conocemos uno de los factores, es fácil computar el segundo.
- Dado un **cifrado de clave pública basado en factorización de números primos**:
 - la **clave pública** contiene un número compuesto de dos factores primos grandes, y **el algoritmo de cifrado** usa ese compuesto para cifrar el mensaje
 - El algoritmo **para descifrar** el mensaje **requiere el conocimiento de los factores primos**, para que el descifrado sea fácil si poseemos la clave privada que contiene uno de los factores, pero extremadamente difícil en caso contrario.

Principales usos de la criptografía de clave pública

- **Cifrado de clave pública**— un mensaje cifrado con la clave pública de un destinatario no puede ser descifrado por nadie, excepto un poseedor de la clave privada correspondiente
 - Se utiliza para confidencialidad de un mensaje
 - Una analogía es un buzón con una ranura de correo. La ranura de correo está expuesta y accesible al público; su ubicación (la dirección de la calle) es, en esencia, la clave pública. Alguién que conozca la dirección de la calle puede ir a la puerta y colocar un mensaje escrito a través de la ranura; sin embargo, sólo la persona que posee la clave puede abrir el buzón de correo y leer el mensaje
- **Firmas digitales**— un mensaje firmado con la clave privada del remitente puede ser verificado por cualquier persona que tenga acceso a la clave pública del remitente, lo que demuestra que el remitente tenía acceso a la clave privada (y por lo tanto, que sea la persona asociada con la clave pública utilizada) y la parte del mensaje que no se ha manipulado
 - Se utiliza para autenticidad de un mensaje
 - Una analogía es el sellado de un sobre con un sello de cera. El mensaje puede ser abierto por cualquier persona, pero la presencia del sello autentifica al remitente

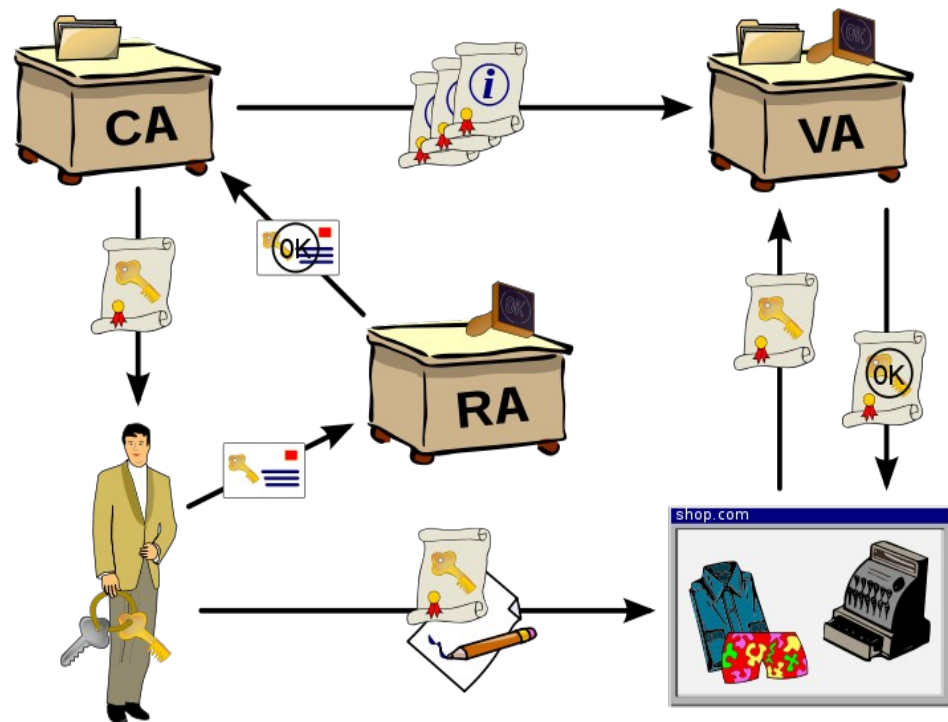
Problema principal del sistema de claves públicas

- El problema principal al usar criptografía asimétrica es saber con seguridad que una **clave pública**:
 - **es correcta**, pertenece a la persona correcta
 - No se ha sido **modificada** o reemplazada por otra persona con fines maliciosos
- Para resolver esto se suele utilizar una **infraestructura de clave pública (PKI)**, sobre la que una autoridad de certificación verifica quien es el dueño de cada clave
 - Una **CA** es una entidad de confianza que confirma la identidad de un solicitante y le envía un certificado
 - **El certificado une la identidad del solicitante a una clave pública**
- Otra manera es la usada por PGP que consiste en utilizar una **"red de confianza"** para autenticar los pares de claves

Infraestructura de clave pública (PKI)

- Consiste en hardware, software, personal, políticas y procedimientos que ayudan a crear, gestionar, distribuir, utilizar, guardar y revocar **certificados digitales**
- En criptografía, un PKI es un acuerdo que une claves públicas a sus respectivos usuarios utilizando una autoridad de certificación (CA)

- Un **usuario solicita un certificado** con su clave pública a una **autoridad de registro (RA)**
- Esta última confirma la identidad del usuario a la **autoridad de certificación (CA)**, la cual emite el certificado
- Ahora el usuario puede firmar un contrato digitalmente utilizando su certificado
- Luego el otro individuo comprueba su identidad utilizando una **autoridad de validación (VA)**, la cual a su vez recibe información sobre certificados emitidos de la CA

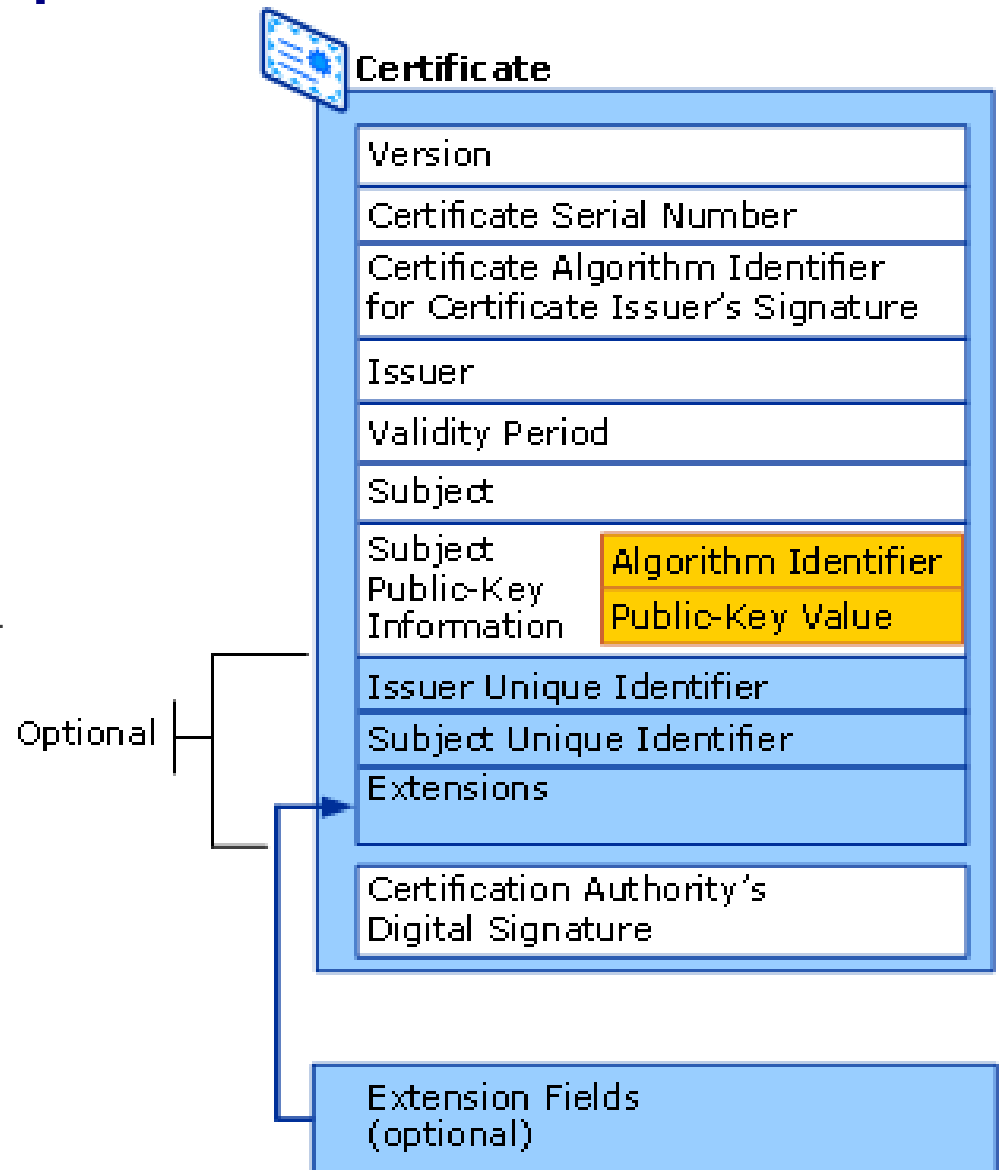


Certificado digital

- Certificado de clave pública o certificado de identidad
- **Es un documento electrónico que utiliza una firma digital para unir una clave pública con una entidad:**
 - Información como el nombre de una persona u organización, su dirección, etc...
- Los certificados digitales funcionan de forma similar a tarjetas de identificación como pasaportes o DNI
- Son emitidos por las **autoridades de certificación (CAs)** que tienen que validar la identidad del titular de un certificado, antes de emitirlo y cuando está en uso.
- Los certificados se pueden usar para verificar que una clave pública pertenece a un individuo
 - Normalmente en una **PKI**, la **firma será de la autoridad de certificación (CA)**
 -
 - En una **red de confianza**, la **firma** es de:
 - Del propio usuario (un **certificado firmado por sí mismo**)
 - U otros usuarios ("**confirmaciones**")
- En cualquier caso, las firmas en los certificados constituyen una garantía por parte de la persona que firma de que la identidad y la clave pública van juntas
- Se pueden crear certificados para servidores basados en Unix con herramientas como **openssl**

Contenidos de un certificado digital X.509 típico

- **Número de serie:** identificador único para el certificado
- **Algoritmo de firma:** El algoritmo usado para crear la firma
- **Emisor:** La entidad que verificó la información y emitió el certificado
- **Válido desde:** La fecha a partir de la cual el certificado tiene validez
- **Válido hasta:** La fecha de caducidad
- **Sujeto:** La persona o identidad identificada
- **Clave pública de sujeto:** El propósito de SSL al usarlo con HTTP no es solo encriptar el tráfico sino también autenticar al propietario de la página web.
- **Uso de clave:** El propósito que tiene la clave pública (e.g. cifrado, firma, firmar otros certificados...)
- **Firma digital de CA:**
 - **Algoritmo de huella:** El algoritmo utilizado para sacar el hash del certificado
 - **Huella:** El hash utilizado para asegurarse de que no se ha modificado el certificado



Jerga: Network sniffing (olfatear, olisquear, husmear)

- Interceptar y registrar el tráfico que pasa a través de una red o parte de ella
- Tiene usos legítimos para monitorear redes con el fin de detectar y analizar fallos o ingeniería inversa de protocolos de red
- También es habitual su uso para fines maliciosos, como robar contraseñas, interceptar mensajes de correo electrónico, espiar conversaciones de chat, etc.
- Véase: `tcpdump` y `wireshark`

Actividad 1:

- Instalar el programa “wireshark”
- hay que ejecutarlo en modo superusuario
- Ver algunos videos y documentación de <http://www.wireshark.org/docs/> para aprender a utilizarlo
- Espiar lo que hace el compañero (en wifi puede ser complicado)
- Poner un filtro para limitarnos al tráfico que venga o vaya a nuestro ordenador
- Arrancar un navegador web (mozilla)
- Entrar en una página https
- Buscar los registros TSL y el certificado del servidor (filtro: ssl.handshake.certificate is present)
- Tiempo estimado: 20 minutos

CA (Certificate Authority) y distribución

- Un PKI X.509 típico permite que cada certificado esté firmado solo por una entidad: la autoridad de certificación (CA)
- El certificado de la propia autoridad puede estar a su vez firmado por otra CA, formando una cadena hasta un certificado raíz que está firmado por sí mismo. Este es el **certificado root**
- Los certificados raíz deben de estar disponibles para aquellos que usan un CA de nivel más bajo por lo que se distribuyen ampliamente
- Se distribuyen con navegadores web, clientes de email, etc...
- De esta manera páginas protegidas con SSL/TLS, mensajes de email, etc... se pueden autenticar sin que los usuarios tengan que instalar certificados raíz manualmente
- Las aplicaciones software normalmente incluyen cientos de certificados raíz de varios PKIs, los que aseguran a su vez la validez de todos los certificados de la jerarquía de la cual forman parte.

Como se crean los certificados

- **Generar un par de claves:** el solicitante genera un par de claves pública y privada
- **Recopilar la información requerida:** el solicitante recopila toda la información que requiere la CA para emitir un certificado
- **Solicitar el certificado:** the applicant sends a certificate request, consisting of his or her public key and the additional required information, to the CA
 - **La solicitud de certificado,** la cual se firma con la clave pública del cliente, **también se puede encriptar utilizando la clave pública de la CA**
- **Verificar la información:** la CA aplica distintas políticas para comprobar si el solicitante debe recibir un certificado
- **Creación del certificado:** la CA crea y firma un documento digital que contiene la clave pública del solicitante y el resto de información pertinente
 - **La firma de la CA** autentica la unión del nombre del sujeto y su clave pública. **El documento firmado es el certificado.**
- **Envío del certificado:** la CA envía el certificado al solicitante o lo coloca en un directorio para que se pueda acceder desde ahí

Validación de la ruta de certificación

- El proceso de validación de la ruta asegura que se pueda establecer una **ruta de certificación** válida para un certificado dado
- Una ruta de certificación válida se define como un certificado de usuario final que se puede seguir a través de una cadena hasta una CA raíz de confianza.
- Hay que encontrar cada CA en la cadena y validarla hasta llegar a la CA raíz.
- Durante el proceso de validación, un certificado puede ser considerado como inválido por muchas razones distintas

Actividad 2: certificados (en Mozilla y linux)

- **Arrancar Mozilla y mirar sus certificados de autoridades instalados:**
 - Editar => Preferencias => Avanzado => Cifrado => Ver Certificados
 - Ponerse sobre una autoridad o servidor y darle al botón de [ver][Detalles]
 - ¿Cuántos campos tiene el certificado de “idp.uca.es”? ¿Quién lo emite?
- Conectar Mozilla con una página segura: por ejemplo, correo de alumnos o “<https://www.bancosantander.es>” y mirar sus certificados:
 - Herramientas => Información de la página => Seguridad => ver certificado => Detalles
 - ¿Quién firma ambos certificados?
 - ¿Cuál es la jerarquía de certificados?
- **Buscar los certificados pre-almacenados en linux:**
 - Esto varía según el sistema operativo pero puedes buscar en /usr/share/ca-certificates
 - Intenta visualizar algunos de los certificados
 - ¿qué son las extensiones crt y pem?
- Tiempo estimado: 20 minutos

Actividad 3: openssl y transmisión segura (1)

- # generando un par de claves publica/privada
- \$ openssl genrsa -out clavesAutoridad.pem
- # visualizando el archivo de claves generado
- \$ openssl rsa -in clavesAutoridad.pem -text
- # Generando un certificado auto-firmado para "autoridad"
- # antes hay que buscar el archivo "openssl.cnf", copiarlo
- # a tu directorio y modificarlo a tu gusto
- \$ openssl req -new -key clavesAutoridad.pem -x509 -out autoridad.pem -config openssl.cnf
-

Actividad 3: openssl y transmisión segura (2)

- # Generando un certificado para "jose" firmado por "autoridad"
- # primero se generan las claves para "jose"
- `$ openssl genrsa -out clavesJose.pem`
- # Ahora generamos un "certificate request" para "jose"
- `$ openssl req -new -key clavesJose.pem -out jose_req.pem -config openssl.cnf`
- # y lo visualizamos
- `$ openssl req -in jose_req.pem -noout -text`

Actividad 3: openssl y transmisión segura (3)

- # ahora "autoridad" va a firmar el "certificate Request" de "jose"
- \$ openssl x509 -req -in jose_req.pem -CA autoridad.pem -CAkey clavesAutoridad.pem -out jose.pem -set_serial 3
- # lo visualizamos
- \$ cat jose.pem
- \$ openssl x509 -in jose.pem -text -noout
- # y, por último, validamos el "Certificate Path"
- \$ openssl verify -CAfile autoridad.pem jose.pem

Actividad 3: openssl y transmisión segura (4)

- Python TLS/SSL socket wrapper
- `ssl.wrap_socket(sock, keyfile=None, certfile=None, server_side=False, cert_reqs=CERT_NONE, ssl_version={see docs}, ca_certs=None, do_handshake_on_connect=True, suppress_ragged_eofs=True, ciphers=None)`

Actividad 3: openssl y transmisión segura (5)

- Usamos los certificados anteriores para una comunicación segura
- El servidor es “**jose**” y tiene que autenticarse
- El cliente no se autentica
- Hay que instalar el certificado de la “autoridad” en el cliente
- Mirar los archivos “**get_put.py**”, “**ssl_client.py**” y “**ssl_server.py**”
- Tiempo estimado: 50 minutos
- **Se pide:**
 - 1) ¿Qué significa la extensión de archivo “pem” ? ¿para qué se utiliza?
 - 2) Modificar cliente y servidor para que el cliente también se autentique
 - 3) Modificar cliente y servidor para que el servidor responda al cliente enviándole su mismo mensaje pero parado a mayúsculas.
 - 4) Utilizar “wireshark” para comprobar que la conexión está encriptada
 - 5) Crear un Certificate-Path con tres que unas firmen a otras y utilizar esta cadena en tu programa

Actividad 4: openssl (encriptar la clave privada)

- Crear un directorio y trabajar dentro
- Generar un par de claves publica/privada
 - `openssl genrsa -out misclaves.pem`
 - Aunque dice que está generando la clave privada, en realidad está generando ambas claves pública/privada y la almacena en formato pem
- Vemos sus componentes:
 - `openssl rsa -in misclaves.pem -text`
- La clave privada es “privada”. Encriptamos las claves (hay que dar una clave):
 - `openssl genrsa -des3 -out misclaves.pem`
- Volvemos a verla encriptada (hay que dar la clave):
 - `openssl rsa -in misclaves.pem -text`
- Tiempo estimado: 2 minutos

Actividad 5: criptografía de clave pública en tus propias palabras

- Ve este vídeo de YouTube
<http://www.youtube.com/watch?v=3QnD2c4Xovk>
- Explica con tus propias palabras qué entiendes por criptografía de clave pública
- Tiempo estimado: 10 minutos



Preguntas de repaso

- Si quieres filtrar las IPs que tienen acceso a tu aplicación, ¿es mejor denegar el acceso desde tu código Python o a nivel de sistema operativo? ¿Por qué?
- ¿Cómo funciona un ataque Man-in-the-Middle si el atacante controla el servidor DNS que usa el cliente?
- Nombra dos usos que tiene la criptografía de clave pública
- ¿Durante la comunicación se encriptan todos los datos normalmente con criptografía de clave pública-privada? ¿Por qué no?
- ¿Qué es una autoridad de certificación (CA)?
- ¿Qué es una ruta de certificación (certification path)?
- ¿Cuál es la diferencia entre SSL y TLS?
- ¿Qué es HTTPS? ¿Contra qué tipo de ataques protege?
- Si encripto un texto con mi clave privada, ¿qué estoy haciendo? ¿y si lo encripto con la clave pública del destinatario?
- ¿Se puede encriptar y firmar un texto a la vez? ¿Qué programa de GNU podrías usar para ello?

TLS & SSL

- **Transport Layer Security (TLS)**
- and its predecessor, **Secure Sockets Layer (SSL)**
- are cryptographic protocols that provide communications security over the Internet
- TLS/SSL enables server authentication, client authentication, data encryption, and data integrity over networks such as the World Wide Web
- TLS and SSL encrypt the segments of network connections above the Transport Layer (TCP), using:
 - **symmetric cryptography** for **privacy**
 - and a **keyed message authentication** code for **message reliability**
- The TLS protocol allows client/server applications to communicate across a network in a way designed to prevent **eavesdropping** and **tampering**

Jargon:

- **Eavesdropping:**
 - is the act of secretly listening to the private conversation of others without their consent
- **Tampering:**
 - deliberate altering or adulteration of information, a product, a package, or system

Jargon: “*reliability*”

- the RFC uses the term "reliable" to refer to two data integrity features:
- The **error detection** provided by TCP (**TCP checksum**)
 - This is a rather weak method:
 - it is possible for the data to become corrupted without the checksum detecting it
 - particularly if there is malicious intent to tamper with a message
- a **keyed-hash MAC (Message Authenticity Code)**, or **HMAC**, to protect the message's data integrity
 - An HMAC algorithm takes a **message** and generates a **hash**
 - Then the **hash is encrypted** with a **secret key**
 - Calculation of the HMAC with the same hash algorithm at the recipient's end of the communication would detect **tampering** with the data
 - And, because the recipient of the message with the MAC also has the secret key, the recipient can verify **authenticity** of the message
 - This means that the message could only have been sent by someone with the same key
 - Do not confuse "authenticity" of the message with authentication of the sender, such as that done with a digital signature

TLS has two layers

- the **TLS Handshake Protocol**
- and the **TLS Record Protocol**
- The **TLS Record Protocol** provides "**reliability**" **separately from** providing **encryption** for several reasons
 - among which is that sometimes only **data integrity** and **authenticity** are required, while **confidentiality** is not
 - and encrypting the data would result in unnecessary overhead
 - The Record Protocol can also be used without encryption

Handshake

- A TLS client and server negotiate a stateful connection by using a handshaking procedure
- During this handshake, client and server agree on various parameters.
 - Handshake begins when client connects to a TLS-enabled server requesting a secure connection and presents a list of supported **CipherSuites (ciphers and hash functions)**.
 - From this list, the server picks the strongest cipher and hash function that it also supports and notifies the client of the decision.
 - The server sends back its identification in the form of a **digital certificate: server name, the trusted certificate authority (CA) and the server's public encryption key**.
 - The client may contact the server that issued the certificate (the trusted CA as above) and confirm the validity of the certificate before proceeding.
 - In order to generate the session keys used for the secure connection, the client **encrypts a random number with the server's public key** and sends the result to the server. Only the server should be able to decrypt it, with its private key.
 - From the random number, both parties generate symmetric key material for encryption and decryption.

- This concludes the handshake and begins the secured connection, which is encrypted and decrypted with the symmetric key material until the connection closes
- If any one of the above steps fails, the TLS handshake fails and the connection is not created
- The **public/private key based encryption is used only for handshaking and symmetric secret keys exchange**
- Once the symmetric secret keys have been exchanged **the symmetric secret keys are used for encryption of the server to client communication**
- This is done because public/private key based encryption techniques are computationally very expensive thus their use should be minimized

- TLS can be used with any client-server application, but our most typical use is with HTTP
- In fact, what is called HTTPS is simply HTTP over TLS (or SSL)
- While the TCP well-known port number used for **HTTP is 80**
- to distinguish secured traffic from insecure traffic, **HTTPS uses port 443**
- Because you implement TLS/SSL beneath the application layer, most of its operations are completely invisible to the client
 - This allows the client to have little or no knowledge of the security of communications and still be protected from attackers.
- There are **a few limitations** to using TLS/SSL, including:
 - **Increased processor load**: cryptography, specifically public key operations, is CPU-intensive
 - **Administrative overhead**: A TLS/SSL environment is complex and requires maintenance; the system administrator must configure the system and manage certificates

Common TLS/SSL Scenarios

- Many people think of TLS and SSL as protocols that are used with Web browsers. However, they are also general purpose protocols that can be used whenever authentication and data protection are necessary
- For example, you can use TLS/SSL for:
 - **SSL-secured transactions with an e-commerce Web site**
 - **Authenticated client access to an SSL-secured Web site**
 - **Remote access**
 - **SQL access**
 - **E-mail**
- This is not an exhaustive list. In fact, the ability to access these protocols through Security Service Provider Interface (SSPI) means that you can use them for just about any application
- Many applications are being modified to take advantage of the features of TLS/SSL.

Symmetric-key algorithm

- are a class of algorithms for cryptography that use trivially related, often identical, cryptographic keys for both decryption and encryption etc.
- The keys, in practice, represent a shared secret between two or more parties that can be used to maintain a private information link.
- Other terms for symmetric-key encryption are single-key, shared-key, one-key
- The problem is to be able to communicate the shared key without any third parties obtaining it

Public-key cryptography

- unlike symmetric key algorithms, a public key algorithm does not require a secure initial exchange of one or more secret keys between the sender and receiver
- involves the use of asymmetric key algorithms:
 - the public key, needed to transform the message to a secure form, is different from the the private key needed to reverse the process
 - The person who anticipates receiving messages first creates both a public key and an associated private key, and publishes the public key
 - When someone wants to send a secure message to the creator of these keys, the sender encrypts it using the intended recipient's public key
 - to decrypt the message, the recipient uses the private key.

The algorithm used for encrypting and decrypting was designed in such a way that:

- while it is easy for the intended recipient to generate the public and private keys
- and to decrypt the message using the private key
- and while it is easy for the sender to encrypt the message using the public key
- it is extremely difficult for anyone to figure out the private key based on their knowledge of the public key.

Theoretical basis:

- Public-key cryptographic systems are based on **one-way and trapdoor functions** that take advantage of certain properties, for example prime numbers.
- **One-way functions** are those whose computation is easy while their inverse operations are extremely hard
 - For example, it's easy to multiply two prime numbers to obtain a composite number, however it's hard to factorize a composite number into its prime components
- A **trapdoor function** is similar, but it has a trap. If you know certain information, it is possible to perform the inverse operation easily.
 - For example, if we have a composite number with two prime factors and we know one of them, it's easy to obtain the second one.
- Given a **public-key cryptographic system based on the factorization of prime numbers**:
 - The **public key** contains a composite number of two large prime factors, and the encryption algorithm uses this number to encrypt the message
 - The algorithm to **decrypt** the message requires knowledge of the **prime factors** for the decryption, it's easy if we know one of the factors, but extremely hard if we don't.

Main uses of public key cryptography

- **Public key cryptography** – a message encrypted with the public key of a receiver cannot be decrypted by anyone else, except the owner of the corresponding private key
 - It's used to keep a message secret
 - An analogy would be a mailbox. The slot is accessible to the public; its location (street address) is, in essence, the public key. Someone who knows the street address can go to this mailbox and place a message inside. However, only the owner of the key can open the mailbox and read the message
- **Digital signatures** --- a signed message with the private key of the sender can be verified by anyone who has access to the public key of this sender, which proves that the person who created it had access to the private key (therefore, he's the person associated to the used public key) and the message hasn't been tampered with.
 - Used for message authentication
 - An analogy would be a wax stamp on an envelope. The message can be opened by anyone, but the stamp authenticates the sender

Problems with public key cryptography

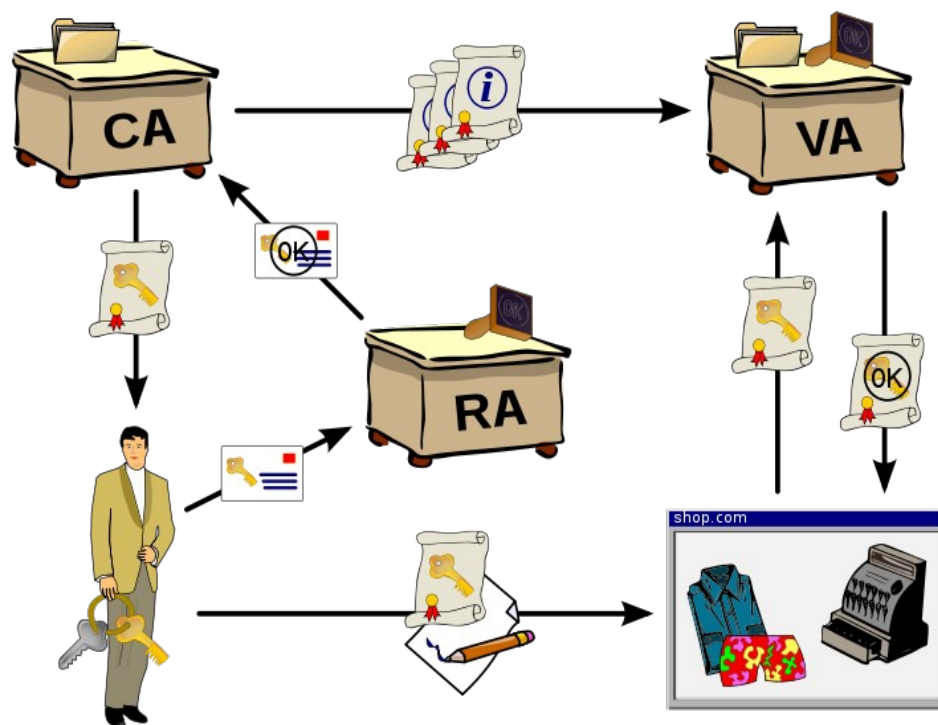
- A central problem for use of public-key cryptography is **confidence** (ideally proof) that **a public key**:
 - **is correct**, belongs to the person or entity claimed (i.e., is 'authentic')
 - and **has not been tampered with** or replaced by a malicious third party
- The usual approach to this problem is to use a **public-key infrastructure (PKI)**, in which one or more third parties, known as **certificate authorities (CA)**, certify ownership of key pairs
 - A **CA** is a mutually-trusted third party that confirms the identity of a certificate requestor (usually a user or computer), and then **issues the requestor a certificate**
 - **The certificate binds the requestor's identity to a public key**
- Another approach, used by PGP, is the "**web of trust**" method to ensure authenticity of key pairs

Public Key Infrastructure (PKI)

Infraestructura de clave pública

- Is a set of hardware, software, people, policies, and procedures needed to create, manage, distribute, use, store, and revoke **digital certificates**
- In cryptography, a PKI is an arrangement that binds public keys with respective user identities by means of a certificate authority (CA)

- A **user applies for a certificate** with his public key at a **registration authority (RA)**
- The latter confirms the user's identity to the **certification authority (CA)** which in turn issues the certificate
- The user can then digitally sign a contract using his new certificate
- His identity is then checked by the contracting party with a **validation authority (VA)** which again receives information about issued certificates by the certification authority.

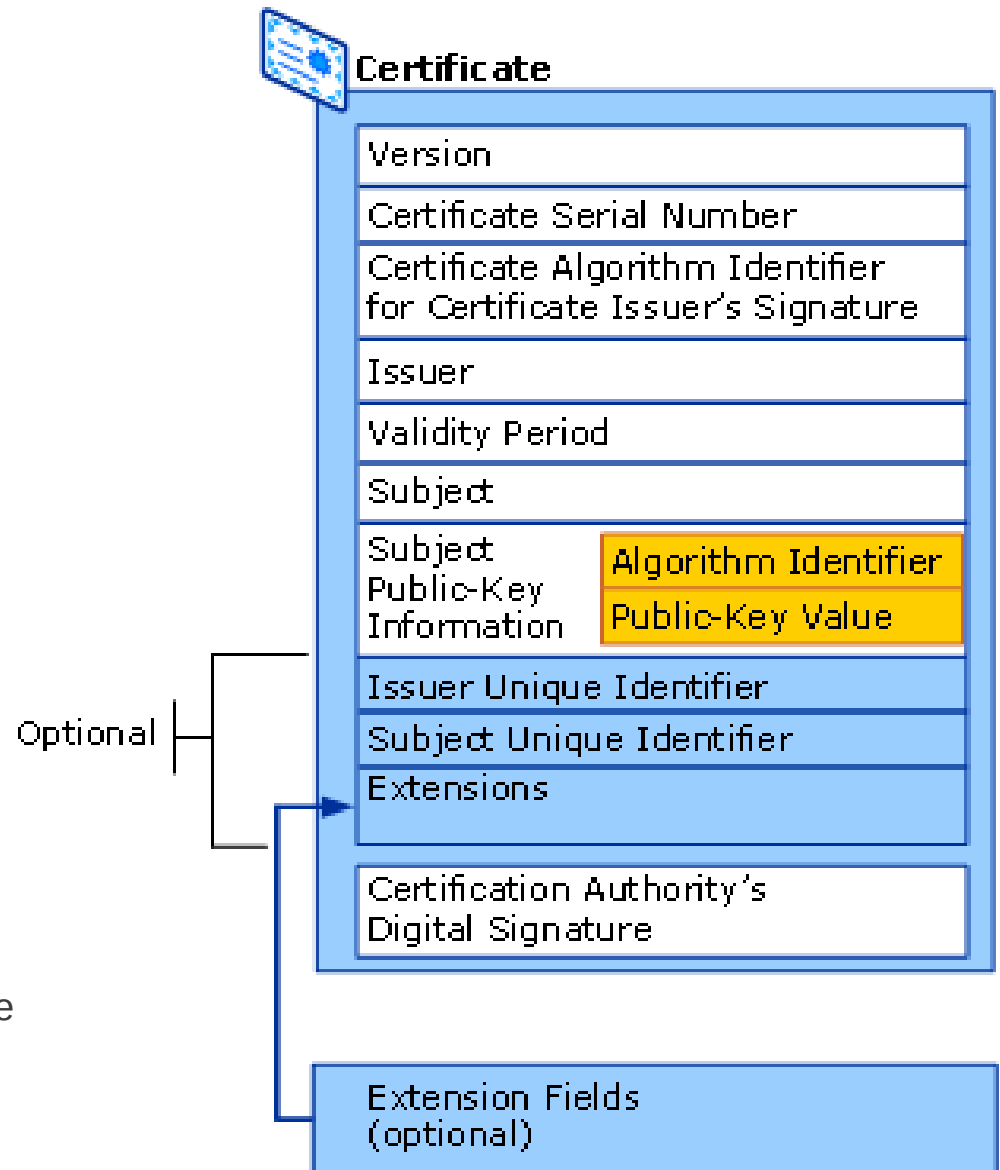


Public key certificate

- **Is an electronic document which uses a digital signature to bind a public key with an identity:**
 - information such as the name of a person or an organization, their address, and so forth
- Digital certificates function similarly to identification cards such as passports and drivers licenses
- They are issued by **certification authorities (CAs)** that must validate the identity of the certificate-holder: both before the certificate is issued, and when the certificate is use
- The certificate can be used to verify that a public key belongs to an individual
 - In a typical **PKI** scheme, the **signature will be of a certificate authority (CA)**
 - In a **web of trust (red de confianza)** scheme, the **signature is of:**
 - either the user (a **self-signed certificate**)
 - or other users ("**endorsements**")
- In either case, the signatures on a certificate are attestations by the certificate signer that the identity information and the public key belong together
- Certificates can be created for Unix-based servers with tools such as OpenSSL's **openssl**

Contents of a typical X.509 digital certificate

- **Serial Number:** uniquely identify the certificate
- **Signature Algorithm:** The algorithm used to create the signature
- **Issuer:** The entity that verified the information and issued the certificate
- **Valid-From:** The date the certificate is first valid from
- **Valid-To:** The expiration date
- **Subject:** The person, or entity identified
- **Subject Public Key:** the purpose of SSL when used with HTTP is not just to encrypt the traffic, but also to authenticate who the owner of the website is
- **Key-Usage:** Purpose of the public key (e.g. encipherment, signature, certificate signing...)
- **CA Digital Signature:**
 - **Thumbprint Algorithm:** The algorithm used to hash the certificate
 - **Thumbprint:** The hash itself to ensure that the certificate has not been tampered with



Network sniffing

- Intercept and log traffic passing over a digital network or part of a network.
- It has legitimate uses like monitoring networks in order to troubleshoot errors or to reverse engineer protocols
- It's also used frequently with malicious intent, like stealing passwords, intercepting emails, eavesdrop on chat conversations, etc...
- See: `tcpdump` and `wireshark`

Exercise 1:

- Install “wireshark”
- You must run it as superuser
- Watch videos and read the docs to learn how to use it:
<http://www.wireshark.org/docs/>
- Spy what your classmate is doing (using wifi might make it difficult)
- Add a filter to limit traffic to your own computer
- Run a web browser (mozilla firefox)
- Visit a web page that uses https
- Look for the TSL registers and the server's certificate (filter: ssl.handshake.certificate is present)
- Estimated time : 20 minutes

CA (Certificate Authority) y distribución

- a typical X.509 PKI permits each certificate to be signed only by a single party: a certificate authority (CA)
- The CA's certificate may itself be signed by a different CA, all the way up to a **'self-signed' root certificate**
- Root certificates must be available to those who use a lower level CA certificate and so are typically distributed widely.
- They are for instance, distributed with such applications as browsers and email clients
- In this way SSL/TLS-protected Web pages, email messages, etc. can be authenticated without requiring users to manually install root certificates
- Applications commonly include over one hundred root certificates from dozens of PKIs, thus by default bestowing trust throughout the hierarchy of certificates which lead back to them

How Certificates Are Created

- **Generate a key pair**: the applicant generates a public and private key pair
- **Collect required information**: The applicant collects whatever information the CA requires to issue a certificate
- **Request the certificate**: the applicant sends a certificate request, consisting of his or her public key and the additional required information, to the CA
 - The certificate request, which is signed with the client's public key, **can also be encrypted by using the CA's public key**
- **Verify the information**: the CA applies whatever policy rules it requires to verify that the applicant should receive a certificate
- **Create the certificate**: the CA creates and signs a digital document containing the applicant's public key and other appropriate information
 - The **signature of the CA** authenticates the binding of the subject's name to the subject's public key. **The signed document is the certificate.**
- **Send or post the certificate**: the CA sends the certificate to the applicant or posts the certificate in a directory, as appropriate.

Certificate path validation

- The path validation process ensures that a valid **certification path** can be established for a given end certificate
- A valid certification path is defined as an end-user certificate that can be traced along a certificate chain to a trusted root CA
- Each CA certificate in the path must be discovered and subsequently validated until a final authority such as a root CA is obtained
- During the validation process, a certificate can be deemed invalid (or not trusted) for many reasons

Exercise 2:certificates (on Mozilla and Linux)

- **Run Mozilla and check its installed certificates:**
 - Edit => Preferences => Advanced => Encryption => View Certificates
 - Select a server and press [view][Details]
 - How many fields does the certificate “idp.uca.es” have? Who sends it?
- Visit a secure page with Mozilla, for example, student email or “<https://www.bancosantander.es>” and check its certificate:
 - Tools => Page Info => Security => View Certificate=> Details
 - Who signs both certificates?
 - What's the certificate hierarchy?
- **Search for the certificates that come with Linux:**
 - This varies depending on the OS but you can check on /usr/share/ca-certificates
 - Try to visualize some of them
 - What are the extensions crt and pem for?
- Estimated time: 20 minutes

Exercise 3: openssl and secure transmission (1)

- # generate a pair of public/private keys
- `$ openssl genrsa -out authorityKeys.pem`
- # visualize the generated file with the keys
- `$ openssl rsa -in authorityKeys.pem -text`
- # Generate a self-signed certificate for "authority"
- # before you must search for the file "openssl.cnf" and copy it to your folder
- # after modifying it to suit your needs
- `$ openssl req -new -key authorityKeys.pem -x509 -out authority.pem -config openssl.cnf`
-

Exercise 3: openssl and secure transmission (2)

- # Generate a certificate for "jose" signed by "authority"
- # first generate the keys for jose
- \$ openssl genrsa -out joseKeys.pem
- # Now generate a certificate request for jose
- \$ openssl req -new -key joseKeys.pem -out jose_req.pem -config openssl.cnf
- # and visualize it
- \$ openssl req -in jose_req.pem -noout -text

Exercise 3: openssl and secure transmission (3)

- # now "authority" will sign jose's certificate request
- `$ openssl x509 -req -in jose_req.pem -CA authority.pem -CAkey authorityKeys.pem -out jose.pem -set_serial 3`
- # visualize it
- `$ cat jose.pem`
- `$ openssl x509 -in jose.pem -text -noout`
- # and last, validate the certificate path
- `$ openssl verify -CAfile authority.pem jose.pem`

Exercise 3: openssl and secure transmission (4)

- Python TLS/SSL socket wrapper
- `ssl.wrap_socket(sock, keyfile=None, certfile=None, server_side=False, cert_reqs=CERT_NONE, ssl_version={see docs}, ca_certs=None, do_handshake_on_connect=True, suppress_ragged_eofs=True, ciphers=None)`

Exercise 3: openssl and secure transmission (5)

- Use certificates for a secure transmission
- The server is “**jose**” and has to be authenticated
- The client doesn't have to be authenticated
- The “authority” certificate must be installed on the client
- Check out the files “**get_put.py**”, “**ssl_client.py**” and “**ssl_server.py**”
- Estimated time: 50 minutes
- **Questions:**
 - 1) **What does the extension “pem” mean? What's it used for?**
 - 2) **Modify client and server so the client is also authenticated**
 - 3) **Modify client and server to allow the server to respond to the client with the same message it receives in upper case**
 - 4) **Using “wireshark” ensure that the connection uses encryption**
 - 5) **Create a Certificate-Path with 3 elements that sign each other and use this path in your program**

Exercise 4: openssl (encrypt the private key)

- Create a new folder
- Generate a pair of public/private keys
 - `openssl genrsa -out mykeys.pem`
 - Even though it says it is generating a private key, in reality it generates both (private and public) and stores them with the pem format
- Let's check its components:
 - `openssl rsa -in mykeys.pem -text`
- The password is “*private*”. We encrypt the keys (you must enter the password):
 - `openssl genrsa -des3 -out mykeys.pem`
- We see it again encrypted (you must provide the password):
 - `openssl rsa -in mykeys.pem -text`
- Estimated time: 2 minutes

Exercise 5: public key cryptography in your own words

- Watch this YouTube video
<http://www.youtube.com/watch?v=3QnD2c4Xovk>
- Explain in your own words what you understand by public key cryptography
- Estimated time: 10 minutes



Review questions

- If you want to limit the Ips that have access to your application, is it better to deny access from your Python code or at the OS level? Why?
- How does a man-in-the-middle attack work if the attacker controls the DNS server used by the client?
- Name two uses of public-key cryptography
- During communication, is every message encrypted using public-key cryptopgrahy? Why not?
- What's a certificate authority (CA)?
- What's a certification path?
- What's the difference between SSL and TLS?
- What's HTTPS? Against which types of attack does it offer protection?
- If I encrypt a message with my private key, what am I doing? What if I encrypt it with the public key of the receiver?
- Can you encrypt and sign data at the same time? What GNU program could you use for that?

Strings en Python

```
>>> s = "cañón"
>>> s
'ca\xc3\xb1\xc3\n'
>>> type(s)
<type 'str'>
>>> len(s)
7
>>> t = u"cañón"
>>> t
u'ca\xf1\xf3n'
>>> type(t)
<type 'unicode'>
>>> len(t)
5
```

- En **Python**, las cadenas de caracteres normales (`str`) tienen forma de secuencia de octetos.
- Además existe la cadena Unicode que **no** la tiene.
- ¿secuencia de caracteres?

En TCP todo se transmite como un stream de octetos

```
>>> s.encode('utf8')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position
2: ordinal not in range(128)
>>> s.encode('latin1')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position
2: ordinal not in range(128)
```

```
>>> t.encode("utf8")
'ca\xc3\xb1\xc3\xb3n'
>>> t.encode('latin1')
'ca\xfa\xfbn'
```

```
>>> type(t)
<type 'unicode'>
>>> type(t.encode("utf8"))
<type 'str'>
```

- Para transmitir datos hay que tener en cuenta la codificación.


```
>>> sDec = s.decode('utf8')
```

```
>>> type(sDec)
```

```
<type 'unicode'>
```

```
>>> len(sDec)
```

```
5
```

```
>>> sDecEnc = sDec.encode('utf8')
```

```
>>> type(sDecEnc)
```

```
<type 'str'>
```

```
>>> len(sDecEnc)
```

```
7
```

```
>>> s
```

```
'ca\xc3\xb1\xc3\xb3n'
```

```
>>> sDecEnc
```

```
'ca\xc3\xb1\xc3\xb3n'
```

```
>>> sDec
```

```
u'ca\x1\x3n'
```

```
>>> s == sDecEnc
```

```
True
```

```
>>> sDec.encode('latin1')
```

```
'ca\xff\x3e'
```

```
>>> sDec.encode('utf16')
```

```
'\xff\xfe\x00a\x00\xff\x00\x3e\x00n\x00'
```

```
>>> sDec.encode('idna')
```

```
'xn--can-8mak'
```

```
>>> sDec.encode('base64')
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
File "/usr/lib/python2.6/encodings/base64_codec.py", line 24, in base64_encode
```

```
    output = base64.encodestring(input)
```

```
File "/usr/lib/python2.6/base64.py", line 315, in encodestring
```

```
    pieces.append(binascii.b2a_base64(chunk))
```

```
UnicodeEncodeError: 'ascii' codec can't encode characters in position 2-3: ordinal not in range(128)
```

```
>>> s.encode('base64')
```

```
'Y2HDscOzbg==\n'
```

```
>>> s.encode('zip')
```

```
'x\x9cKN<\xbc\xff\x00\xe6<\x00\x0f\xb2\x04\x1d'
```

Ejemplo: `tema5_codecs2.py`

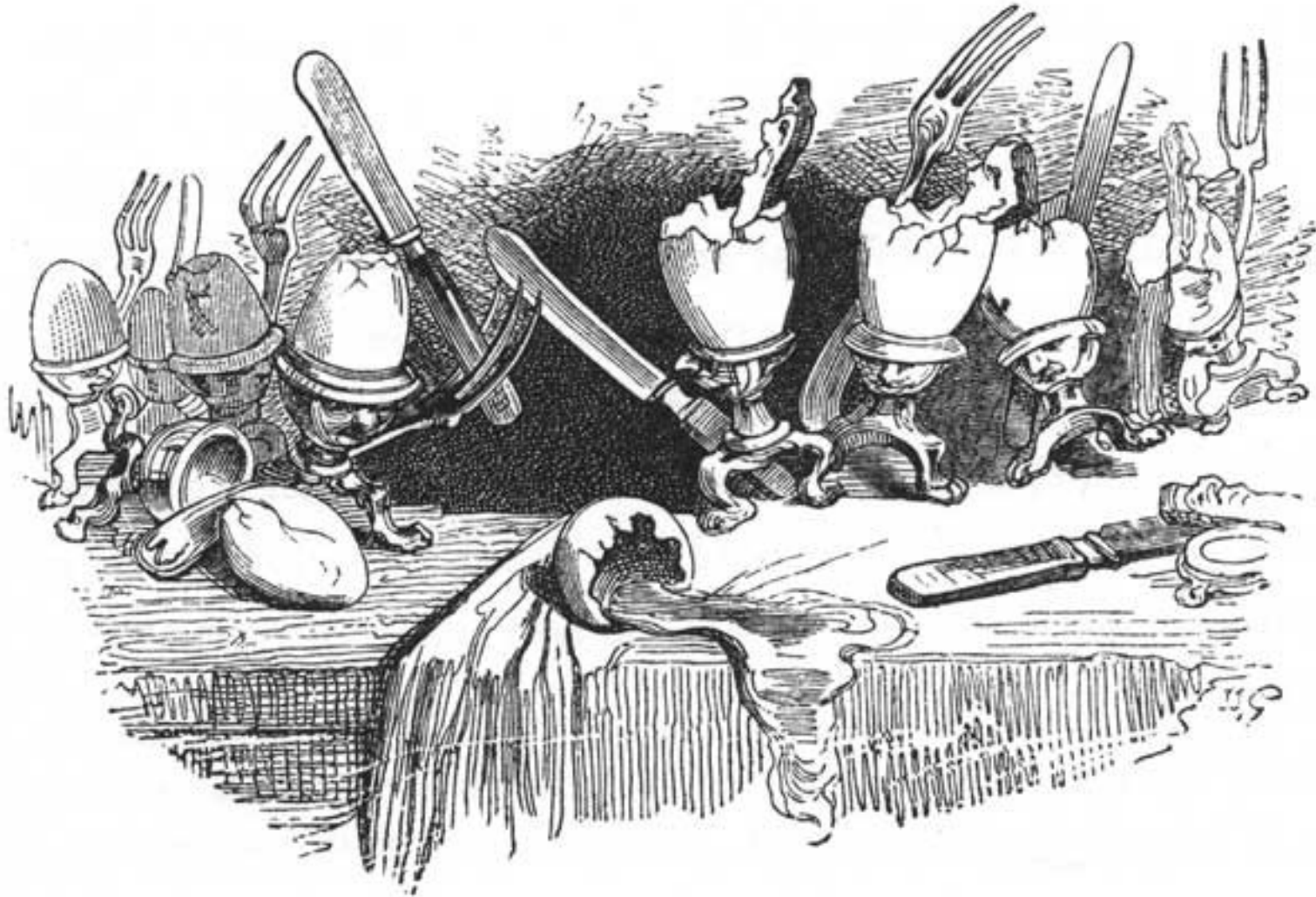
Funciona todo bien salvo '**idna**' que no permite que la cadena entre dos puntos '.' tenga más o igual de 64 caracteres ni tampoco permite que se comience con un punto

Archivo de entrada codificado en UTF-8: `quijote.txt`

para transmitir datos hay que convertirlos en cadenas

- Las cadenas de caracteres se deben **codificar** en alguno de los métodos conocidos: UTF8, UTF16, latin1, etc. y **decodificarlos** a la vuelta.
- Para datos binarios existe el paquete “**struct**” que los empaqueta como cadenas
- Para los enteros (binarios) hay un convenio **big endian-little endian** de transferencia en la red **ntoh()** y **hton()**
 - Network = big-endian
 - En Python se puede hacer con **struct**
- **>>> import struct**
- **>>> struct.pack('>i', 1234567)**
- **'\x00\x12\xd6\x87'**
- **>>> struct.pack('<i', 1234567)**
- **'\x87\xd6\x12\x00'**
- **>>> struct.unpack('<i', struct.pack('<i', 1234567))**
- **(1234567,)**

Matanza en la mesa del desayuno



Este fue el resultado de la guerra en Little-Endians y the Big-Endians.
Grabado de la edition de 1838 de los Viajes de Gulliver. Por J. J. Grandville.

para transmitir otros objetos hay que serializarlos-deserializarlos

- Los datos estructurados (diccionarios, listas, etc.) hay que **serializarlos-deserializarlos**:
 - Paquete “**pickle**” propio de Python
 - **Json**
 - **XML**
- ```
- >>> import json
- >>> a = {'hola':45, 'adios':[1,2,3]}
- >>> type(a)
- <type 'dict'>
- >>> json.dumps(a)
- '{"hola": 45, "adios": [1, 2, 3]}'
- >>> type(json.dumps(a))
- <type 'str'>
- >>> json.loads(json.dumps({'hola':45, 'adios':[1,2,3]}))
- {u'hola': 45, u'adios': [1, 2, 3]}
- >>> type(json.loads(json.dumps({'hola':45, 'adios':[1,2,3]})))
- <type 'dict'>
```

# Enmarcado y Entrecomillado

- En caso de que sólo se transmita **un dato**, la marca de final puede ser el fin del stream (se cierra el socket)
- En caso de realizarse un **diálogo** con varios datos, hay que marcar el final de cada envío “**enmarcado**” (*framing*)
  - Por ejemplo: pickle termina en punto '.'
    - `>>> pickle.dumps(4253)`
    - `'I4253\n.'`
    - `>>> pickle.dumps([4253, 'www.uca.es', u'cañón'])`
    - `"(lp0\nI4253\naS'www.uca.es'\np1\naVca\xff1\xff3n\np2\na."`
    - `>>>`  
`pickle.loads(pickle.dumps([4253, 'www.uca.es', u'cañón']))`
    - `[4253, 'www.uca.es', u'ca\xff1\xff3n']`
  - Los puntos dentro del dato se escapan automáticamente (*quoting*)
  - Otros formatos pueden elegir otros sistemas de *framing* (saltos de línea, llaves, longitud del dato, etc.)

# Distintos patrones: Leer hasta final del stream

- Leer hasta final del stream (es decir, hasta que se cierre el socket)

```
message = ''
while True:
 more = sc.recv(8355)
 if not more:
 break
 message += more
```

- O bien:

```
message = sc.makefile().read()
```



## Otro patrón: leer un número de octetos dado

- Debemos saber de antemano la longitud del mensaje (o bien enviar el mensaje junto con su longitud)

```
def recvall(sock, length):
 data = ''
 while length > len(data):
 bytes_remaining = length - len(data)
 more = sock.recv(bytes_remaining)
 if not more:
 raise EOFError('No se recibieron todos los bytes. Socket cerrado')
 data += more
 return data
```

## Otro patrón: enviamos el mensaje con su longitud

- Al mensaje original le añadimos, al comienzo, su longitud empaquetada con struct

```
def put(sock, message):
 mensajeConLongitud = struct.pack('i', len(message)) + message
 sock.sendall(mensajeConLongitud)
```

- Leemos primero la longitud del mensaje --que son los primeros struct.calcsize('i') bytes-- y luego el mensaje en sí
- Se utiliza a la función “recvall” de la transparencia anterior para leer un mensaje de una cierta longitud

```
def get(sock):
 lendata = recvall(sock, struct.calcsize('i'))
 (length,) = struct.unpack('i', lendata)
 return recvall(sock, length)
```

# Paquete pickle para serializar

- Es propio de python (no interface con Java o C++)
- La ventaja es que tiene enmarcado automático

```
>>> import pickle
>>> pickle.dumps([1,2,3])
'(lp0\nI1\naI2\naI3\na.'
>>> pickle.loads(pickle.dumps([1,2,3]))
[1, 2, 3]

>>> a = pickle.dumps([1,2,3]) + pickle.dumps({"hola":45, "adios":34})
>>> a
'(lp0\nI1\naI2\naI3\na.(dp0\nS'hola'\np1\nI45\nsS'adios'
 \np2\nI34\ns.'"
>>> pickle.loads(a)
[1, 2, 3]
>>> import StringIO
>>> f = StringIO.StringIO(a)
>>> pickle.load(f)
[1, 2, 3]
>>> pickle.load(f)
{'hola': 45, 'adios': 34}
```

# Paquete json para serializar

```
>>> import json
>>> json.dumps([1,2,3])
'[1, 2, 3]'

>>> json.loads(json.dumps([1,2,3]))
[1, 2, 3]

>>> f = StringIO.StringIO(json.dumps([1,2,3]))
>>> json.load(f)
[1, 2, 3]
```

- Problemas del paquete json:
  - No empaqueta binarios
  - No hace un “framing” bueno
- Existen otras alternativas para serializar:
  - Google Protocol Buffer
  - XML

# Preguntas de repaso (FPNP:Capítulo 5)

- Después de haber leído el tema 5 del libro. Comprueba si eres capaz de responder a las siguientes preguntas:
  - ¿Qué significa **idna** y para qué se utiliza dicho código? ¿qué limitaciones tiene?
  - Proporciona dos ejemplos de protocolos que utilicen el código **base64**.
  - Nombra dos formatos de codificación de caracteres con longitud variable y dos con longitud fija. ¿Cuál piensas que es más eficiente en términos de espacio?
  - ¿Qué diferencia existe entre los tipos de cadena: (str) y (unicode) en Python?
  - Si sabemos el tamaño en bytes de una cadena de tipo (unicode) ¿podemos deducir el número de caracteres que tiene? ¿y si es ASCII?
  - ¿Cuáles son las ventajas de enviar datos a través de la red con formato binario? ¿y las desventajas?
  - ¿Qué orden de bytes (Little o Big Endian) es el estándar para la transmisión por red? ¿Es el mismo que utiliza tu ordenador?

# Preguntas de repaso

- Después de haber leído el tema 5 del libro. Comprueba si eres capaz de responder a las siguientes preguntas:
  - ¿Por qué es necesario que en la comunicación a través de red ambas máquinas se pongan de acuerdo sobre la codificación de caracteres?
  - Nombra dos formatos de codificación de caracteres con longitud variable y dos con longitud fija. ¿Cuál piensas que es más eficiente en términos de espacio?
  - Si sabemos el tamaño en bytes de una cadena en UTF-8, ¿podemos deducir el número de caracteres que tiene? ¿y si es ASCII?
  - ¿Es igual el código para la letra 'a' en UTF-8 y ASCII?
  - ¿Cuáles son las ventajas de enviar datos a través de la red con formato binario? ¿y las desventajas?
  - ¿Qué orden de bytes (Little o Big Endian) es el estándar para la transmisión por red? ¿Es el mismo que utiliza tu ordenador?

# Preguntas de repaso

- ¿Para qué sirve el módulo **struct** de Python?
- Durante la comunicación entre dispositivos ¿qué estrategias pueden seguir para que la máquina receptora pueda parar de llamar a la función `recv()` de forma segura sin que se pierdan datos?
- Con respecto a la pregunta anterior ¿qué estrategia piensas que es la mejor? ¿Cuál utiliza el protocolo HTTP?
- ¿Para qué sirve el módulo **pickle** en Python?
- ¿Cuál es el principal inconveniente de JSON? ¿Qué otro formato estándar se puede utilizar para estas situaciones en las que JSON no es adecuado?
- ¿Dónde está el cuello de botella en la transmisión por red? ¿Cómo se puede reducir su impacto?
- ¿Cómo se puede comprimir en Python?
- ¿Cuáles son tres excepciones de red que puede devolver Python?
- ¿Cuál es la manera más correcta de codificar **try...except** para manejar excepciones si estas escribiendo una biblioteca en Python?

# Actividad 1

- **Objetivo:**

- Queremos enviar el primer capítulo del Quijote a través de un socket y protocolo TCP, pero hay que enviarlo codificado.

- **Procedimiento:**

- Localizar una versión accesible electrónicamente del primer capítulo del Quijote.
- Investigar qué tipo de codificación tiene.
- Escriba un **programa servidor** (utilizando socket-TCP) que:
  - Reciba un mensaje de un cliente con un tipo de codificación de las siguientes: **utf-8, base64, idna, latin1, windows-1252, zip.**
  - Lea el primer capítulo del Quijote de un archivo.
  - Lo codifique con la codificación elegida y lo envíe al cliente.
- Escriba un **programa cliente** (utilizando socket-TCP) que:
  - Lea, de la línea de comandos, un tipo de codificación.
  - Envíe un mensaje al servidor con una cadena que indique el tipo de codificación elegido.
  - Reciba, del servidor, el archivo codificado.
  - Lo salve en un archivo.
  - Lo muestre por pantalla.
- Que los programas tengan en cuenta las posibles excepciones.
- Tiempo estimado: 50 minutos
- Nota: al utilizar idna, la cadena no puede tener más de 63 caracteres ni comenzar con punto



# Recursos para Actividad 1

- Los siguientes epígrafes del Capítulo 5 (Network Data and Network Errors) del libro “Foundations of Python ...”
  - “Text and encodings”
  - “Network Byte Order”
  - Compression
  - Network Exceptions
  - Handling Exceptions
- Codificaciones estándar en la “Librería de Referencia” de Python (string service):
  - <http://docs.python.org/library/codecs.html#standard-encodings>
- File Objects en la “Librería de Referencia” de Python:
  - <http://docs.python.org/library/stdtypes.html#file-objects>
- Buscar información sobre qué es “base64” e “idna” en wikipedia

# Actividad 2

- Hay que escribir un servidor y un cliente que se intercambian información mediante un diálogo (sin cerrar el socket hasta el final)
- El servidor creará un diccionario con algunos datos de los alumnos del grupo: nombre, dni, teléfono (por lo menos)
- Se realiza un bucle:
  - El cliente contacta con el servidor enviándole, como mensaje, el nombre de un alumno (método, incluir la longitud al principio)
  - El servidor responde con la ficha del alumno (serializada)
  - Si no existe el nombre se envía algún mensaje de error al cliente
  - El cliente muestra en pantalla los datos recibidos
- El bucle termina, y se cierran los sockets, cuando el cliente envía un mensaje en blanco (longitud 0).
- Hacerlo (incluyendo siempre la longitud del mensaje al principio):
  - Serializada con pickle
  - Serializada con Json
- Tiempo estimado: 40 minutos

# Actividad 2

- Hay que escribir un servidor y un cliente que se intercambian información mediante un diálogo (sin cerrar el socket hasta el final)
- El servidor creará un diccionario con algunos datos de los alumnos del grupo: nombre, dni, teléfono (por lo menos)
- Se realiza un bucle:
  - El cliente contacta con el servidor enviándole, como mensaje, el nombre de un alumno (método, incluir la longitud al principio)
  - El servidor responde con la ficha del alumno (serializada)
  - Si no existe el nombre se envía algún mensaje de error al cliente
  - El cliente muestra en pantalla los datos recibidos
- El bucle termina, y se cierran los sockets, cuando el cliente envía un mensaje en blanco (longitud 0).
- Hacerlo (incluyendo siempre la longitud del mensaje al principio):
  - Serializada con pickle
  - Serializada con Json
- Tiempo estimado: 40 minutos

## Actividad 2: Pistas técnicas

- Modificar el programa “`blocks.py`” del libro (pag. 77)
- Ojo: quitar las dos líneas en que los sockets se cierran en una dirección (tanto en el cliente como servidor)
- El diccionario se puede crear en memoria estático (en el propio código)
- Para leer del teclado se puede utilizar la función: `raw_input([prompt])`

# Actividad 3

Crea un pequeño chat usando TCP en el que un cliente se comunique con un servidor y ambos puedan enviar mensajes.

Los dos deben de introducir un nick al principio. El primer mensaje que se envíe será el nick del cliente al servidor, el servidor responderá con su nick y a partir de este momento podrán chatear.

Para que sea más fácil el ejercicio, la conversación se hará por turnos. Es decir primero envía un mensaje el cliente y luego es el turno del servidor, hasta que este último no responda el cliente no puede mandar otro mensaje.

# Actividad 3: requisitos técnicos

Puedes utilizar `raw_input` para pedir una entrada por teclado

- La comunicación se realizará a través de un único socket que no se cerrará hasta el final del chat.
- Cada mensaje estará almacenado en una estructura tipo “diccionario” de Python: `{“tipo”:tipo, “payload”:mensaje}`
- Los tipos posibles de mensajes son:
  - “nick” al comenzar para intercambiarse los nicks.
  - “adios” para terminar el chat.
  - “normal” para enviar un mensaje.

Todo el mensaje estará serializado utilizando el paquete `Pickle`.

# In TCP everything is transmitted as a stream of octets

```
>>> s = "cañón"
>>> type(s)
<type 'str'>
```

```
>>> len(s)
7
```

```
>>> s.encode('utf8')
```

```
Traceback (most recent call last):
```

```
 File "<stdin>", line 1, in <module>
```

```
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in
position 2: ordinal not in range(128)
```

```
>>> s.encode('latin1')
```

```
Traceback (most recent call last):
```

```
 File "<stdin>", line 1, in <module>
```

```
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in
position 2: ordinal not in range(128)
```

Character strings already have the form of an octet sequence.  
The only thing to take into account is the encoding.

```
>>> sDec = s.decode('utf8')
>>> type(sDec)
<type 'unicode'>

>>> len(sDec)
5

>>> sDecEnc = sDec.encode('utf8')
>>> type(sDecEnc)
<type 'str'>

>>> len(sDecEnc)
7

>>> s
'ca\xc3\xb1\xc3\xb3n'

>>> sDecEnc
'ca\xc3\xb1\xc3\xb3n'

>>> sDec
u'ca\xf1\xf3n'

>>> s == sDecEnc
True
```



```
>>> sDec.encode('latin1')
```

```
'ca\xff\x3e'
```

```
>>> sDec.encode('utf16')
```

```
'\xff\xfe\x00a\x00\xff\x00\x3e\x00n\x00'
```

```
>>> sDec.encode('idna')
```

```
'xn--can-8mak'
```

```
>>> sDec.encode('base64')
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
File "/usr/lib/python2.6/encodings/base64_codec.py", line 24, in base64_encode
```

```
 output = base64.encodestring(input)
```

```
File "/usr/lib/python2.6/base64.py", line 315, in encodestring
```

```
 pieces.append(binascii.b2a_base64(chunk))
```

```
UnicodeEncodeError: 'ascii' codec can't encode characters in position 2-3: ordinal not in range(128)
```

```
>>> s.encode('base64')
```

```
'Y2HDscOzbg==\n'
```

```
>>> s.encode('zip')
```

```
'x\x9cKN<\xbc\xff\x00\xe6<\x00\x0f\xb2\x04\x1d'
```

Ejemplo: `tema5_codecs2.py`

Example: `tema5_codecs2.py`

In this example everything works correctly except 'idna' encoding, which doesn't allow a string between two dots to have more than 63 characters and it can't begin with a dot.

Input file encoded as UTF-8: `quijote.txt`

# Problems in data transmission:

- Character strings must be **encoded / decoded** with a known format: UTF8, UTF16, latin1, etc...
- For binary data you can use the module “**struct**” in Python which packs the data as strings
- Integer data is usually transmitted as big endian over a network. You can use the functions `ntoh()` y `hton()`
- Data structures such as dictionaries and lists have to be **serialized / deserialized**
  - Use Python's module “**pickle**”
  - **Json**
  - **XML**

# More problems in data transmission:

- In case that data is only transmitted once, to signal the end of the transmission the end of the stream itself can be used (when the socket is closed)
- When there is data going back and forth, the end of each transmission must be signaled somehow, this is called “framing”

- For example, Python's pickle ends with a dot '.'

```
- >>> pickle.dumps(4253)
- 'I4253\n.'
- >>> pickle.dumps([4253, 'www.uca.es', u'cañón'])
- "(lp0\nI4253\naS'www.uca.es'\np1\naVca\xfa\xfb\nnp2\na."
- >>> pickle.loads(pickle.dumps([4253, 'www.uca.es', u'cañón']))
- [4253, 'www.uca.es', u'ca\xfa\xfb']
```

- Dots are escaped automatically
- Different formats can use different mechanisms for framing (new lines, braces, data length, etc.)

```
- >>> json.dumps([4253, 'www.uca.es', u'cañón'])
- '[4253, "www.uca.es", "ca\\u00fa\\u00fb"]'
```

## Different patterns: Read to the end of the stream

- Read to the end of the stream (i.e. Until the socket is closed)

```
message = ''
while True:
 more = sc.recv(8355)
 if not more:
 break
 message += more
```

- Or:

```
message = sc.makefile().read()
```

## A different pattern: read a given number of octets

- We know beforehand the message length or we could send the message length together with the message itself

```
def recvall(sock, length):
 data = ''
 while length > len(data):
 bytes_remaining = length - len(data)
 more = sock.recv(bytes_remaining)
 if not more:
 raise EOFError('No se recibieron todos los bytes. Socket
cerrado')
 data += more
 return data
```

## Another pattern: send the message with its length

- We embed the length of the message at the start of the original message. This length is stored with struct

```
def put(sock, message):
 messageWithLength = struct.pack('i', len(message)) + message
 sock.sendall(messageWithLength)
```

- First we read the length of the message (which is stored in the first `struct.calcsize('i')` bytes), afterwards we can read the message itself
- You can use the function “`recvall`” from the previous slide to read a message with a given length

```
def get(sock):
 lendata = recvall(sock, struct.calcsize('i'))
 (length,) = struct.unpack('i', lendata)
 return recvall(sock, length)
```

# **pickle** module for serialiazation

- Used in python (no interface with Java or C++)

```
>>> import pickle

>>> pickle.dumps([1,2,3])
'(lp0\nI1\naI2\naI3\na.'
```

```
>>> pickle.loads(pickle.dumps([1,2,3]))
[1, 2, 3]
```

```
>>> import StringIO

>>> f = StringIO.StringIO(pickle.dumps([1,2,3])
 +pickle.dumps(("hola que tal", 5)))

>>> pickle.load(f)
[1, 2, 3]
```

```
>>> pickle.load(f)
('hola que tal', 5)
```



# json module for serialization

```
>>> import json
>>> json.dumps([1,2,3])
'[1, 2, 3]'

>>> json.loads(json.dumps([1,2,3]))
[1, 2, 3]

>>> f = StringIO.StringIO(json.dumps([1,2,3]))
>>> json.load(f)
[1, 2, 3]
```

- Some problems with json are:
  - It can't be used for binary data
  - It doesn't frame well
- Alternatives for serialization
  - Google Protocol Buffer
  - XML

# Review questions (FPNP: Chapter 5)

Why is it necessary during network transmission that both ends agree on which encoding to use?

Name two character encodings with variable character length and two with fixed length. Which do you think is most efficient in terms of space?

If we know the length in bytes of a UTF-8 string, can we deduce the number of characters it contains? And if it were an ASCII string?

Is the code for the letter 'a' the same in UTF-8 and ASCII?

What are the advantages and disadvantages of sending data over the network in binary format?

Which byte order (Little or Big Endian) is the standard for network transmissions? Is it the same one your computer uses?

# Review questions

What's the module **struct** used for in Python?

When two devices communicate with each other, what strategies can they both follow so the receiving end knows when to stop calling the function **recv()** in a safe without losing any data?

Regarding question 8, which strategy do you consider to be the best one?  
Which one does HTTP use?

What's **pickle** used for in Python?

What's the main disadvantage of using JSON? Which other standard format can you use for these situations in which JSON can't do the job?

Where is the bottleneck during network transmission? How can it be reduced?

How can you compress data in Python?

Which are the three network exceptions that Python can throw?

Which is the more correct way to write **try...except** to handle exceptions when you are writing a library in Python?

# Exercise 1

- **Goal**

- We want to send the first chapter of Don Quixote through a socket using TCP, but it has to be encoded

- **How-to:**

- Grab an online copy of the first chapter of Don Quixote
  - Find out what encoding it uses
  - Write a **server** (using TCP-socket) that:
    - Receives a message from a client with an encoding type from the following ones: **utf-8**, **base64**, **idna**, **latin1**, **windows-1252**, **zip**.
    - Reads the first chapter of Don Quixote from a file
    - The server should send it to the client with the encoding he chose

- Write a **client** (using TCP-socket) that:
    - Reads an encoding type from the command line
    - Sends a string to the server containing the chosen type of encoding.
    - Receives the encoded file from the server.
    - Saves it in a local file
    - Shows it on screen

- The programs should handle exceptions

- Estimated time: 50 minutes

- Note: when using idna, the string cannot be longer than 63 characters

# Resources of Exercise 1

- The following section from Chapter 5 (Network Data and Network Errors) of the book “Foundations of Python ...”
  - “Text and encodings”
  - “Network Byte Order”
  - Compression
  - Network Exceptions
  - Handling Exceptions
- Standard codecs from the Python standard library:
  - <http://docs.python.org/library/codecs.html#standard-encodings>
- File Objects from the Python standard library:
  - <http://docs.python.org/library/stdtypes.html#file-objects>
- Look for information on “base64” and “idna” on Wikipedia

# Exercise 2

- You must write a server and a client that exchange information through a dialog (without closing the socket till the end of the transmission)
- The server will build a dictionary with data about students from the group: name, ID number, phone number, etc..
- There should be a loop
  - The client contacts the server sending it as a message the name of the student (prepend message length to each message)
  - The server responds with a serialized record of the student
  - If there isn't a record for a given name, the server sends back an error
  - The client shows on screen the received data
- The loop ends and the sockets are closed when the client sends an empty message (length 0 bytes)
- Do it prepending the message length before each message:
  - Using pickle to serialize
  - Using json to serialize
- Estimated time: 40 minutes

## Exercise 2: hints

- Modify the code from “`blocks.py`” shown in the book (p. 77)
- Watch out: remove the lines in which the socket is closed in one direction (both client and server)
- The dictionary with the data can be created in memory and be written directly in the source file
- To read from standard input you can use:  
`raw_input( [prompt] )`

# Exercise 3

- a) Create a small chat using TCP where a client communicates with a server and either one can send messages. Users at both ends must introduce a nickname when the program starts. The first message sent will be a nickname from the client to the server, the server will reply with its own nickname and from then on they will be able to chat. In order to make it easier for you, the conversation will be carried out in turns. In other words, first the client sends a message, then it's the server's turn, until the server replies the client can't send another message.
- b) Modify the previous chat program to accept Unicode emoticons. When a person writes :), it must be replaced by ☺ . Visit the following link to see a list of Unicode emoticons:  
[www.unicode.org/charts/PDF/U2600.pdf](http://www.unicode.org/charts/PDF/U2600.pdf)
- Estimated time: 40 minutes for a) and 10 minutes for b)



# Exercise 3: hints

- Look at the code from the book on page 77-78 and copy a modified version of the loop used in the server into the client
- When someone talks on the chat, it will become the other person's turn. Therefore even though the loops will be similar, design them in such a way that the client always talks first. If the loops are exactly the same and both wait at the beginning for the other one to talk, then there will be a lock
- Remove the lines where the sockets are shutdown. They should be closed at the end
- Use `raw_input()` to get a nickname from the users

# Nombres de los sockets

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- Prototipo en C:
- Cuando **creamos** un socket hay que fijar 3 parámetros:
- **Familia de direcciones (domain)**. Determina el tipo de red: internet, Unix, AppletTalk, Bluetooth, ...
  - Típicamente AF\_INET
- **Tipo de socket**. Determina la técnica de comunicación que vamos a utilizar
  - Aunque cada familia puede tener sus propios tipos, estos tienden a ser independientes de las familias
  - SOCK\_DGRAM (basado en paquetes)
  - SOCK\_STREAM (basado en flujo y con control de fiabilidad)
- **Protocolo**.
  - Suele haber una única opción una vez fijada la familia de direcciones y el tipo de socket. Por eso suele dejarse en blanco (a cero) para que se elija automáticamente
  - En nuestro caso: TCP y UDP

# Address Families en

## /usr/include/bits/socket.h

*/\* Address families. \*/*

```
#define AF_UNSPEC PF_UNSPEC
#define AF_LOCAL PF_LOCAL
#define AF_UNIX PF_UNIX
#define AF_FILE PF_FILE
#define AF_INET PF_INET
#define AF_AX25 PF_AX25
#define AF_IPX PF_IPX
#define AF_APPLETALK PF_APPLETALK
#define AF_NETROM PF_NETROM
#define AF_BRIDGE PF_BRIDGE
#define AF_ATMPVC PF_ATMPVC
#define AF_X25 PF_X25
#define AF_INET6 PF_INET6
#define AF_ROSE PF_ROSE
#define AF_DECnet PF_DECnet
#define AF_NETBEUI PF_NETBEUI
#define AF_SECURITY PF_SECURITY
#define AF_KEY PF_KEY
```

```
#define AF_NETLINK PF_NETLINK
#define AF_ROUTE PF_ROUTE
#define AF_PACKET PF_PACKET
#define AF_ASH PF_ASH
#define AF_ECONET PF_ECONET
#define AF_ATMSVC PF_ATMSVC
#define AF_RDS PF_RDS
#define AF_SNA PF_SNA
#define AF_IRDA PF_IRDA
#define AF_PPPOX PF_PPPOX
#define AF_WANPIPE PF_WANPIPE
#define AF_LLC PF_LLC
#define AF_CAN PF_CAN
#define AF_TIPC PF_TIPC
#define AF_BLUETOOTH PF_BLUETOOTH
#define AF_IUCV PF_IUCV
#define AF_RXRPC PF_RXRPC
#define AF_ISDN PF_ISDN
#define AF_PHONET PF_PHONET
#define AF_IEEE802154 PF_IEEE802154
#define AF_MAX PF_MAX
```

# Tipos de socket en `/usr/include/bits/socket.h`

```
/* Types of sockets. */
enum __socket_type
{
 SOCK_STREAM = 1, /* Sequenced, reliable, connection-based byte streams. */
#define SOCK_STREAM SOCK_STREAM
 SOCK_DGRAM = 2, /* Connectionless, unreliable datagrams of fixed maximum length. */
#define SOCK_DGRAM SOCK_DGRAM
 SOCK_RAW = 3, /* Raw protocol interface. */
#define SOCK_RAW SOCK_RAW
 SOCK_RDM = 4, /* Reliably-delivered messages. */
#define SOCK_RDM SOCK_RDM
 SOCK_SEQPACKET = 5, /* Sequenced, reliable, connection-based, datagrams of
 fixed maximum length. */
#define SOCK_SEQPACKET SOCK_SEQPACKET
 SOCK_DCCP = 6, /* Datagram Congestion Control Protocol. */
#define SOCK_DCCP SOCK_DCCP
 SOCK_PACKET = 10, /* Linux specific way of getting packets at the dev level. For writing
 rarp and other similar things on the user level. */
#define SOCK_PACKET SOCK_PACKET

 /* Flags to be ORed into the type parameter of socket and socketpair and
 used for the flags parameter of paccept. */

 SOCK_CLOEXEC = 02000000, /* Atomically set close-on-exec flag for the new descriptor(s). */
#define SOCK_CLOEXEC SOCK_CLOEXEC
 SOCK_NONBLOCK = 04000 /* Atomically mark descriptor(s) as non-blocking. */
#define SOCK_NONBLOCK SOCK_NONBLOCK
};
```

# Address Families y tipos de socket en Python:

```
import socket; dir(socket)
```

```
'AF_APPLETALK',
'AF_ASH',
'AF_ATMPVC',
'AF_ATMSVC',
'AF_AX25',
'AF_BLUETOOTH',
'AF_BRIDGE',
'AF_DECnet',
'AF_ECONET',
'AF_INET',
'AF_INET6',
'AF_IPX',
'AF_IRDA',
'AF_KEY',
```

```
'AF_LLC',
'AF_NETBEUI',
'AF_NETLINK',
'AF_NETROM',
'AF_PACKET',
'AF_PPPOX',
'AF_ROSE',
'AF_ROUTE',
'AF_SECURITY',
'AF_SNA',
'AF_TIPC',
'AF_UNIX',
'AF_UNSPEC',
'AF_WANPIPE',
'AF_X25'
```

```
'SOCK_DGRAM',
'SOCK_RAW',
'SOCK_RDM',
'SOCK_SEQPACKET',
'SOCK_STREAM'
```

# Python: protocolos para la Familia AF\_INET

'IPPROTO\_AH',  
'IPPROTO\_DSTOPTS',  
'IPPROTO\_EGP',  
'IPPROTO\_ESP',  
'IPPROTO\_FRAGMENT',  
'IPPROTO\_GRE',  
'IPPROTO\_HOPOPTS',  
'IPPROTO\_ICMP',  
'IPPROTO\_ICMPV6',  
'IPPROTO\_IDP',  
'IPPROTO\_IGMP',

'IPPROTO\_IP',  
'IPPROTO\_IPIP',  
'IPPROTO\_IPV6',  
'IPPROTO\_NONE',  
'IPPROTO\_PIM',  
'IPPROTO\_PUP',  
'IPPROTO\_RAW',  
'IPPROTO\_ROUTING',  
'IPPROTO\_RSVP',  
**'IPPROTO\_TCP'**,  
'IPPROTO\_TP',  
**'IPPROTO\_UDP'**

# Protocolos para la familia AF\_INET

## en el archivo: /usr/include/netinet/in.h

```
/* Standard well-defined IP protocols. */
enum
{
 IPPROTO_IP = 0, /* Dummy protocol for TCP. */
#define IPPROTO_IP IPPROTO_IP

 IPPROTO_IPIP = 4, /* IPIP tunnels (older KA9Q tunnels use 94) */
#define IPPROTO_IPIP IPPROTO_IPIP
 IPPROTO_TCP = 6, /* Transmission Control Protocol. */
#define IPPROTO_TCP IPPROTO_TCP
 IPPROTO_EGP = 8, /* Exterior Gateway Protocol. */
#define IPPROTO_EGP IPPROTO_EGP
 IPPROTO_PUP = 12, /* PUP protocol. */
#define IPPROTO_PUP IPPROTO_PUP
 IPPROTO_UDP = 17, /* User Datagram Protocol. */
#define IPPROTO_UDP IPPROTO_UDP

 IPPROTO_IPV6 = 41, /* IPv6 header. */
#define IPPROTO_IPV6 IPPROTO_IPV6

};
```

# Dirección

- Una vez fijados la **familia, tipo y protocolo**
- Cuando queremos conectar un socket hay que proporcionar más parámetros
- En el caso de **IPv4**, 2 parámetros extra:
  - **Host**: entero de 32 bits =  $4 \times 8$  bits
  - **Puerto**: entero de 16 bits
- En el caso de **IPv6**, 4 parámetros extra:
  - **Host**: entero de 128 bits =  $8 \times 16$  bits
  - **Puerto**: entero de 16 bits
  - **Flowinfo (difusión)** unicast, anycast, multicast
  - **Scopeid (alcance)**. En qué parte de la red es válida la dirección: enlace, global, etc
- En otros casos el número de parámetros extra puede variar



# Dirección IP y ARP

- Sabéis que la dirección IP es la que se utiliza en el protocolo IP para enrutamiento (routing)
- Aunque en última instancia (ordenadores en la misma red local) se manejan direcciones físicas MAC
- existe un servicio **ARP** (Address Resolution Protocol) y **RARP** que mapea direcciones IP a direcciones físicas **MAC** (*Media Access Control Address*), y a la inversa
- ARP maneja un caché de direcciones físicas de la subred.
- En Unix hay unos comandos “arp” y “rarp” para manejar este caché.
- Si hay que conectar con un ordenador que no está en la caché se envía un broadcast llamado **ARP request frame**.
- “**arp**” se utiliza cuando se conoce IP pero no la MAC.
- “**rarp**” (reverse) cuando se conoce la MAC pero no la IP.
- No obstante, podemos manejarnos sólo con la IP (es transparente).

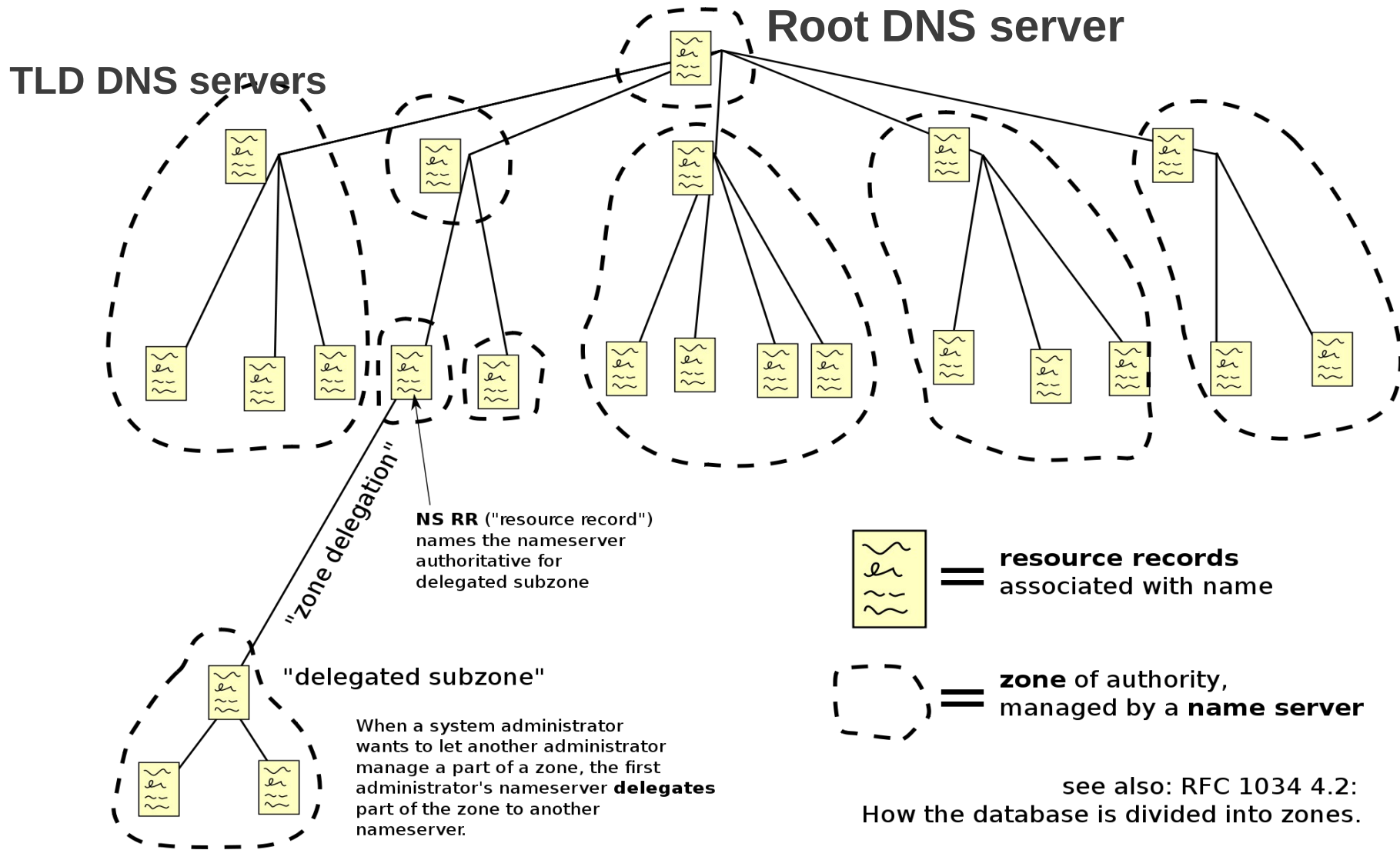
# Dirección IP

- A menudo, lo que conocemos es el nombre del host: por ejemplo “**www.uca.es**”
- Para buscar la dirección IP de un nombre hace falta un servicio de nombres DNS (*Domain Name System*)
- DNS es una base de datos distribuida y jerárquica que almacena información asociada a nombres de dominio
- DNS asocia diferentes tipos de información a cada nombre
- Los usos más comunes son la asignación de nombres de dominio a direcciones IP y la localización de los servidores de correo electrónico de cada dominio.
- DNS utiliza tres componentes principales:
  - **Clientes DNS:** genera peticiones DNS de resolución de nombres a un servidor DNS
  - **Servidores DNS:** Los servidores recursivos tienen la capacidad de reenviar la petición a otro servidor si no disponen de la dirección solicitada
  - Y las **Zonas de autoridad**, porciones del espacio de nombres de dominio que almacenan los datos
    - Cada zona de autoridad abarca al menos un dominio y posiblemente sus subdominios

# DNS

- **TLD (Top Level Domains):** net, org, gov, uk, es ...posible sufijos para nombres de dominio válidos
  - Cada TLD tiene sus propios servidores y organización que lo gestiona
- **Nombre de dominio:** nombres que las organizaciones añaden a sus hosts como python.org, uca.es, etc.
  - Tienes que pagar anualmente pero te da derecho a tener todos los hosts que quieras en dicho dominio
- **Fully qualified domain name (FQDN):** nombre completo de un host. Suele consistir de el nombre del ordenador unido al nombre del dominio
- **Hostname:**
  - Es un término ambiguo. A veces se refiere al FQDN y otras veces al nombre resumido del ordenador sin el dominio
  - FQDN identifican a un ordenador desde cualquier lugar
  - El nombre resumido sólo permite acceder dentro de la organización

# Domain Name Space



# Registro de Recurso en DNS Zone File

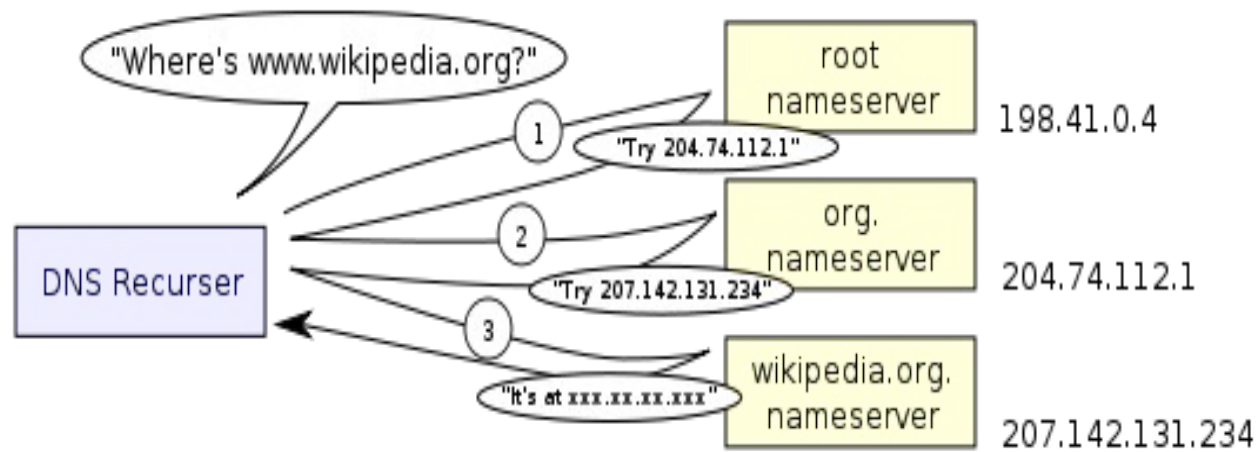
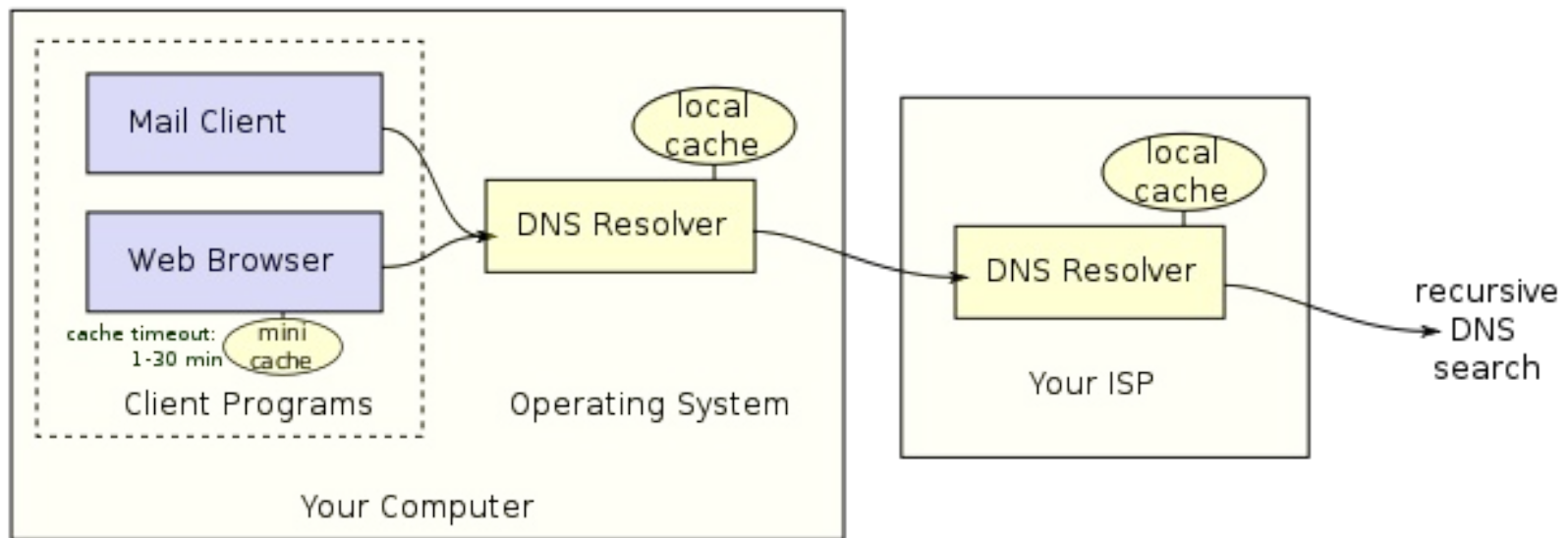
| Descripción |                                                                                     | Longitud (octetos) |
|-------------|-------------------------------------------------------------------------------------|--------------------|
| NAME (fqdn) | Name of the node to which this record pertains                                      | (variable)         |
| TYPE        | Tipo de RR en forma numérica e.g. 15 para MX RRs)                                   | 2                  |
| CLASS       | Código de clase                                                                     | 2                  |
| TTL         | Número de segundos que RR es válido (El máximo es 2 <sup>31</sup> -1, unos 68 años) | 4                  |
| RDLLENGTH   | Longitud del campo RDATA                                                            | 2                  |
| RDATA       | Datos específicos de RR adicionales                                                 | (variable)         |

# Tipo de Registro de Recurso (algunos)

| Type  | Description                       | Function                                                                                                     |
|-------|-----------------------------------|--------------------------------------------------------------------------------------------------------------|
| A     | Registro de dirección             | Devuelve una dirección IPv4 de 32 bits, normalmente usada para asociar hostnames a una dirección IP de host  |
| AAAA  | Registro de dirección IPv6        | Devuelve una IPv6 address de 128 bits, normalmente usada para asociar hostnames a la dirección IP de el host |
| CERT  | Registro de certificado           | Almacena PKIX, SPKI, PGP, etc.                                                                               |
| CNAME | Registro de nombre canónico       | Alias de un nombre a otro. La búsqueda DNS continuará intentándolo con el nombre nuevo.                      |
| DNAME | Nombre de delegación              | DNAME crea un alias para un nombre y todos sus subnombres                                                    |
| LOC   | Registro de lugar                 | Especifica la situación geográfica asociada con un nombre de dominio                                         |
| MX    | Registro de intercambio de correo | Asocia un nombre de dominio a una lista de agentes de transferencia de mensaje para ese dominio              |
| NS    | Registro de servidores de nombres | Delega una zona DNS para que use unos servidores de nombres dados                                            |
| PTR   | Registro Puntero                  | Apunta a el nombre canónico. El uso más común es para implementar búsquedas DNS inversas                     |

# Para buscar una dirección a partir de un nombre, el S.O.

- Se busca localmente en los archivos **/etc/hosts** ó **/windows/system32/drivers/hosts**
- También mantiene una caché de búsquedas anteriores:
- Si no lo encuentra (o la entrada en el caché está caducada), entonces envía mensajes del **protocolo dns** a los servidores que aparecen en el archivo **/etc/resolv.conf** que suelen ser servidores de nombres de nuestro dominio
- Obviamente, esto cambia para cada sistema operativo y cada configuración
- Los servidores de nombre de nuestro dominio, buscan en su base de datos (de su dominio) y en su propia caché (por si ha habido búsquedas análogas anteriores)
- Cada entrada en la caché tienen una fecha de caducidad
- Si no se encuentra, se inicia una búsqueda jerárquica en los servidores de TLD (las direcciones de estos son fijas)
- Se reenvían la búsqueda a los servidores de nombres de los subdominios
- Todo esto suele consumir bastante tiempo



**Los  
gráficos  
están  
sacados  
de  
Wikipedia**



# Ejemplos de búsqueda

```
jose@salmon:~$ nslookup www.uca.es
```

```
Server: 10.141.1.3
```

```
Address: 10.141.1.3#53
```

Véanse los números de puerto predefinidos:

```
Name: www.uca.es
```

```
Address: 150.214.86.11
```

[http://en.wikipedia.org/wiki/List\\_of\\_TCP\\_and\\_UDP\\_port\\_numbers](http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers)

Archivos: `/etc/resolv.conf` y `/etc/hosts` en ubuntu

```
Generated by NetworkManager
```

```
domain uca.es
```

```
search uca.es
```

```
nameserver 10.141.1.3
```

```
nameserver 150.214.76.3
```

```
nameserver 150.214.76.26
```

```
127.0.0.1 localhost
```

```
127.0.1.1 salmon
```

```
The following lines are desirable for IPv6 capable hosts
```

```
::1 localhost ip6-localhost ip6-loopback
```

```
fe00::0 ip6-localnet
```

```
ff00::0 ip6-mcastprefix
```

```
ff02::1 ip6-allnodes
```

```
ff02::2 ip6-allrouters
```

```
ff02::3 ip6-allhosts
```

## Archivo: /windows/system32/drivers/etc/hosts de windows XP

```
Copyright (c) 1993-1999 Microsoft Corp.
#
Éste es un ejemplo de archivo HOSTS usado por Microsoft TCP/IP para Windows.
#
Este archivo contiene las asignaciones de las direcciones IP a los nombres de
host. Cada entrada debe permanecer en una línea individual. La dirección IP
debe ponerse en la primera columna, seguida del nombre de host correspondiente.
La dirección IP y el nombre de host deben separarse con al menos un espacio.
#
#
Por ejemplo:
#
102.54.94.97 rhino.acme.com # servidor origen
38.25.63.10 x.acme.com # host cliente x

127.0.0.1 localhost

127.0.0.1 activate.adobe.com
127.0.0.1 practivate.adobe.com
127.0.0.1 ereg.adobe.com
127.0.0.1 activate.wip3.adobe.com
127.0.0.1 wip3.adobe.com
127.0.0.1 3dns-3.adobe.com
127.0.0.1 3dns-2.adobe.com
127.0.0.1 adobe-dns.adobe.com
127.0.0.1 adobe-dns-2.adobe.com
127.0.0.1 adobe-dns-3.adobe.com
127.0.0.1 ereg.wip3.adobe.com
127.0.0.1 activate-sea.adobe.com
127.0.0.1 wwis-dubc1-vip60.adobe.com
127.0.0.1 activate-sjc0.adobe.com
```

jose@salmon:~\$ **dig flikr.com ANY**

Domain Information Gopher  
Para consultar informaciónb de dominio

```
;; <<>> DiG 9.7.0-P1 <<>> flikr.com ANY
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 29029
;; flags: qr rd ra; QUERY: 1, ANSWER: 7, AUTHORITY: 5, ADDITIONAL: 0

;; QUESTION SECTION:
;flikr.com. IN ANY

;; ANSWER SECTION:
flikr.com. 20943 IN A 68.180.206.184
flikr.com. 20943 IN A 206.190.60.37
flikr.com. 172143 IN NS ns4.yahoo.com.
flikr.com. 172143 IN NS ns5.yahoo.com.
flikr.com. 172143 IN NS ns1.yahoo.com.
flikr.com. 172143 IN NS ns2.yahoo.com.
flikr.com. 172143 IN NS ns3.yahoo.com.

;; AUTHORITY SECTION:
flikr.com. 172143 IN NS ns3.yahoo.com.
flikr.com. 172143 IN NS ns4.yahoo.com.
flikr.com. 172143 IN NS ns5.yahoo.com.
flikr.com. 172143 IN NS ns1.yahoo.com.
flikr.com. 172143 IN NS ns2.yahoo.com.

;; Query time: 1 msec
;; SERVER: 10.141.1.3#53(10.141.1.3)
;; WHEN: Thu Mar 31 10:36:07 2011
;; MSG SIZE rcvd: 225
```

```
jose@salmon:~jose$ dig -x 10.141.9.175

; <<>> DiG 9.7.0-P1 <<>> -x 10.141.9.175
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 58215
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1,
ADDITIONAL: 1

;; QUESTION SECTION:
;175.9.141.10.in-addr.arpa. IN PTR

;; ANSWER SECTION:
175.9.141.10.in-addr.arpa. 172800 INPTR salmon2.uca.es.

;; AUTHORITY SECTION:
141.10.in-addr.arpa. 172800 IN NS baltasar.uca.es.

;; ADDITIONAL SECTION:
baltasar.uca.es.172800 IN A 150.214.76.3

;; Query time: 0 msec
;; SERVER: 10.141.1.3#53(10.141.1.3)
;; WHEN: Thu Mar 31 12:07:13 2011
;; MSG SIZE rcvd: 110
```

```
jose@salmon:~$ whois -d -B -h whois.ripe.net 150.214.86.11
% This is the RIPE Database query service.
% The objects are in RPSL format.
%
% Information related to '150.214.0.0 - 150.214.255.255'

inetnum: 150.214.0.0 - 150.214.255.255
netname: CICA
descr: Centro Informatico Cientifico de Andalucia
descr: Sevilla
country: ES
. . . .
% Information related to '150.214.0.0/16AS766'
route: 150.214.0.0/16
descr: RICA
origin: AS766
mnt-by: REDIRIS-NMC
. . . .
% Information related to '214.150.in-addr.arpa'
domain: 214.150.in-addr.arpa
descr: Centro Informatico Cientifico de Andalucia
descr: Av. de Reina Mercedes SN
descr: Sevilla, 41012
admin-c: SBG9-RIPE
tech-c: SBG9-RIPE
zone-c: SBG9-RIPE
nserver: dns1.cica.es
nserver: dns2.cica.es
nserver: sun.rediris.es
nserver: chico.rediris.es
nserver: ns.ripe.net
. . . .
```

# Mejor utilizar los servicios del S.O.

- Como vemos, la estructura de una dirección depende de la familia: Ipv4, Ipv6, AppleTalk, ...
- Además, cuando buscamos una dirección es conveniente utilizar los recursos del S.O. (primero el archivo hosts y caché, luego los servidores de nombres de nuestro dominio y, sólo en caso necesario, iniciar una búsqueda jerárquica)
- **Python** proporciona la función: **`socket.getaddrinfo()`** que utiliza al sistema operativo, es capaz de manejar muchas familias de direcciones, y se puede utilizar de muchas formas diferentes
- De todas maneras, a veces queremos hacer una búsqueda en los servidores DNS por nosotros mismos

# `socket.getaddrinfo(host, port, family=0, socktype=0, proto=0, flags=0)`

- La mejor forma de manejar todos estos parámetros es a través de la función `getaddrinfo()`
- Traduce el argumento host/puerto a una secuencia de tupla de 5 elementos que contiene todos los arguments necesarios para **crear un socket** conectado a ese servicio.
- **Host** es un nombre de dominio, una cadena que representa una dirección IPv4/6 o None.
- **Port** es una cadena con el nombre de un servicio como “http”, un puerto o None
- Si pasamos None como el valor de host y puerto, puedes pasar NULL a la **API de C de capas inferiores**
- Los argumentos de **familia**, **tipo de socket** y **protocolo** pueden ser especificados **opcionalmente** para reducir la lista de direcciones devueltas.
- Pasar cero como cualquiera de estos argumentos, selecciona todo el rango de resultados.
- El argumento de **flags** (banderines) puede ser una o varias de las constantes `AI_*`, e influirá como se computan los resultados y se devuelven.
  - For example, `AI_NUMERICHOST` will disable domain name resolution and will raise an error if host is a domain name.
  - Por ejemplo, `AI_NUMERICHOST` deshabilitará la resolución de nombres de dominio y lanzará un error si un host es un nombre dominio.

# `socket.getaddrinfo(host, port, family=0, socktype=0, proto=0, flags=0)`

- La función devuelve una tupla de 5 elementos con los siguientes valores:
  - **(family, socktype, protocol, canonname, sockaddr)**
- En estas tuplas, familia, tipo de socket, protocolo son enteros y **se deben pasar a** la función **socket()**.
- canonname será una cadena que representa el nombre canónico del host if `AI_CANONNAME` es parte del argumento `flags`; sino canonname estará vacío.
- sockaddr es una tupla que describe una dirección de socket, cuyo formato depende de la familia devuelta:
  - (address, port) 2-tuple for `AF_INET`
  - (address, port, flow info, scope id) 4-tuple for `AF_INET6`
  - and **is meant to be passed** to the **socket.connect()** method.
- El siguiente ejemplo toma información de la dirección para una conexión TCP a `www.python.org` en el puerto 80 (los resultados pueden cambiar en su sistema si IPv6 no está habilitado):
- 

```
>>> socket.getaddrinfo("www.python.org", 80, 0, 0,
 socket.SOL_TCP)
[(2, 1, 6, '', ('82.94.164.162', 80)),
 (10, 1, 6, '', ('2001:888:2000:d::a2', 80, 0, 0))]
```



# Ejemplos para crear un socket y conectarlo

El siguiente ejemplo utiliza `socket.getaddrinfo()` para crear un socket y conectarlo al puerto HTTP de “gatech.edu”.

```
>>> import socket
>>> infolist = socket.getaddrinfo('gatech.edu', 'www')
>>> infolist
[(2, 1, 6, '', ('130.207.244.244', 80)),
 (2, 2, 17, '', ('130.207.244.244', 80))]
```

La respuesta indica que hay dos formas de crear el socket:

- 1) Tipo `SOCK_STREAM=1` y protocolo `IPPROTO_TCP=6`
- 2) Tipo `SOCK_DGRAM=2` y protocolo `IPPROTO_UDP=17`  
(oficialmente, HTTP soporta TCP y UDP)

```
>>> ftpca = infolist[0]
>>> ftpca # Family Type Protocol Canonical-name Address
(2, 1, 6, '', ('130.207.244.244', 80))
>>> ftpca[0:3]
(2, 1, 6)
>>> s = socket.socket(*ftpca[0:3])
>>> ftpca[4]
('130.207.244.244', 80)
>>> s.connect(ftpca[4])
```

# Más ejemplos de `getaddrinfo()`: en el lado del servidor:

- Queremos crear un socket pasivo en el servidor para servir tráfico “smtp” utilizando “tcp”:

```
>>> i = socket.getaddrinfo(None, 'smtp', 0, socket.SOCK_STREAM, 0, socket.AI_PASSIVE)
>>> i
[(2, 1, 6, '', ('0.0.0.0', 25)), (10, 1, 6, '', (':::', 25, 0, 0))]
>>> ftpca = i[0] # Family Type Protocol Canonical-name Address
>>> ftpca
(2, 1, 6, '', ('0.0.0.0', 25))
>>> FamTypProto = ftpca[0:3]
>>> FamTypProto
(2, 1, 6)
>>> s = socket.socket(*FamTypProto)
```

- Y ahora queremos crear un socket pasivo en el servidor para tráfico “dns” utilizando “udp”:

```
>>> j = socket.getaddrinfo(None, 53, 0, socket.SOCK_DGRAM, 0, socket.AI_PASSIVE)
>>> j
[(2, 2, 17, '', ('0.0.0.0', 53)), (10, 2, 17, '', (':::', 53, 0, 0))]
>>> s2 = socket.socket(*j[0][0:3])
```

# Extracto de <http://www.iana.org/assignments/port-numbers> y de “/etc/services”

## PORT NUMBERS

(last updated 2011-03-29)

The port numbers are divided into three ranges: the Well Known Ports, the Registered Ports, and the Dynamic and/or Private Ports.

The Well Known Ports are those from 0 through 1023.

. . . . .

Port Assignments:

| Keyword       | Decimal       | Description                    | References |
|---------------|---------------|--------------------------------|------------|
| -----         | -----         | -----                          | -----      |
|               | 0/tcp         | Reserved                       |            |
|               | 0/udp         | Reserved                       |            |
| #             |               | Jon Postel <postel&isi.edu>    |            |
| spr-itunes    | 0/tcp         | Shirt Pocket netTunes          |            |
| spl-itunes    | 0/tcp         | Shirt Pocket launchTunes       |            |
| . . . . .     |               |                                |            |
| #             |               | Rick Adams <rick&UUNET.UU.NET> |            |
| <b>smtp</b>   | <b>25/tcp</b> | <b>Simple Mail Transfer</b>    |            |
| <b>smtp</b>   | <b>25/udp</b> | <b>Simple Mail Transfer</b>    |            |
| . . . . .     |               |                                |            |
| <b>domain</b> | <b>53/tcp</b> | <b>Domain Name Server</b>      |            |
| <b>domain</b> | <b>53/udp</b> | <b>Domain Name Server</b>      |            |
| . . . . .     |               |                                |            |

## Más ejemplos de `getaddrinfo()`: cliente y servidor

- Ahora suponemos que un cliente conectarse con un servidor concreto conocido: `www.uca.es` por el puerto “http” utilizando una familia dada (INET) y tipo “SOCK\_STREAM”.
- O bien un servidor quiere conectarse con el “localhost” por el puerto 1060, utilizando la familia INET6 y el protocolo “udp”:

```
>>> i = socket.getaddrinfo('www.uca.es', 'http', socket.AF_INET, socket.SOCK_STREAM)
>>> i
[(2, 1, 6, '', ('150.214.86.11', 80))]
>>> s = socket.socket(*i[0][0:3])
>>> s.connect(i[0][4])

>>> j = socket.getaddrinfo('localhost', 1060, socket.AF_INET6, 0, socket.IPPROTO_TCP)
>>> j
[(10, 1, 6, '', ('::1', 1060, 0, 0))]
>>> ftpca = j[0] # Family Type Protocol Canonical-name Address
>>> k = socket.socket(*ftpca[0:3])
>>> a = j[0][4] # Address
>>> a
('::1', 1060, 0, 0)
>>> k.bind(a)
```

# Ejercicio de getaddrinfo ( )

¿Qué formas de hay de conectarse con “[www.uca.es](http://www.uca.es)” ?

## Más ejemplos de getaddrinfo(): flags

```
>>> i = socket.getaddrinfo('iana.org', 'www', 0, socket.SOCK_STREAM, 0,
 socket.AI_ADDRCONFIG | socket.AI_V4MAPPED)
>>> i
[(2, 1, 6, '', ('192.0.43.8', 80))]
```

```
>>> i = socket.getaddrinfo('iana.org', 'www', 0, socket.SOCK_STREAM, 0,
 socket.AI_ADDRCONFIG | socket.AI_V4MAPPED | socket.AI_CANONNAME)
>>> i
[(2, 1, 6, '43-8.any.icann.org', ('192.0.43.8', 80))]
```

- **AI\_ADDRCONFIG** filtra las direcciones que no son posibles desde tu ordenador
- **AI\_V4MAPPED** devuelve las direcciones Ipv4 que puedes utilizar pero recodificadas como direcciones Ipv6 (caso de que tu máquina tenga interfaz Ipv6 pero el servicio que tu necesitas sólo utiliza Ipv4)
- **AI\_CANONNAME** realiza una búsqueda DNS y devuelve el nombre canónico del host
- Por último, **AI\_CANONNAME**. Un servidor encuentra el nombre canónico del cliente que ha conectado con él

```
>>> sc, (direcc, puerto) = s.accept()
>>> i = socket.getaddrinfo(direcc, puerto, sc.family, sc.type,
 sc.proto, socket.AI_CANONNAME)
>>> print "i =", i
i = [(2, 1, 6, 'rr.pmta.wikimedia.org', ('208.80.152.2', 80))]
```

# Ejemplo de utilización en un programa

- El programa “**www\_ping.py**” utiliza la información devuelta por **getaddrinfo()** para ver si un equipo permite conectarse por tcp
- Algo parecido a “ping”
- La utilización de **getaddrinfo()** nos permite escribir un programa totalmente general e independiente de la familia: Ipv4, Ipv6, AppleTalk, etc. y del protocolo

```
jose@salmon:~$./www_ping.py mit.edu
Éxito: host WEB.MIT.EDU está escuchando por el puerto 80
jose@salmon:~$./www_ping.py smtp.google.com
fallo de la red: Connection timed out
jose@salmon:~$./www_ping.py ordenador-inexistente.com
Fallo del servicio de nombres: No address associated with hostname
```

# Otro ejemplo más de utilización

- El programa “**forward\_reverse.py**”
- Busca la dirección IP de un nombre y luego, a la inversa, busca el nombre canónico de esa dirección
- Por último, compara ambos nombres (directo e inverso)

```
jose@salmon:~$./forward_reverse.py flickr.com
flickr.com has IP address 68.142.214.24
68.142.214.24 has the canonical name www.flickr.vip.mud.yahoo.com
WARNING! The forward and reversenames belongs to a different organization
jose@salmon:~$./forward_reverse.py mit.edu
mit.edu has IP address 18.9.22.69
18.9.22.69 has the canonical name WEB.MIT.EDU
The forward and reverse names have a lot in common
jose@salmon:~$./forward_reverse.py www.uca.es
www.uca.es has IP address 150.214.86.11
150.214.86.11 has the canonical name www.uca.es
Estupendo, los nombres coinciden completamente
```



# Paquete pydns

- `pip install pydns`
- Sirve para realizar, a mano, búsquedas DNS
- Ejemplo de utilización: `dns_basic.py`

```
jose@salmon:~$./dns_basic.py python.org
python.org IN A '82.94.164.162'
python.org IN AAAA ' \x01\x08\x88 \x00\x00\r\x00\x00\x00\x00\x00\x00\xa2 '
python.org IN MX (50, 'mail.python.org')
python.org IN NS 'ns2.xs4all.nl'
python.org IN NS 'ns.xs4all.nl'
jose@salmon:~$./dns_basic.py uca.es
uca.es IN A '150.214.86.11'
uca.es IN MX (40, 'melchor.uca.es')
uca.es IN MX (10, 'merlin.uca.es')
uca.es IN MX (12, 'saruman.uca.es')
uca.es IN MX (15, 'horus.uca.es')
uca.es IN MX (20, 'smtp1.uca.es')
uca.es IN NS 'dns1.cica.es'
uca.es IN NS 'dns2.cica.es'
uca.es IN NS 'nimue.uca.es'
uca.es IN NS 'baltasar.uca.es'
```

# RFC 5321

- Este documento especifica las reglas para resolver dominios de e-mail:
- Es caso de existir registros de tipo MX debes contactar con dichos servidores SMTP y devolver un error al usuario si ninguno de ellos aceptan el mensaje
- En caso de no existir registros MX pero si existir registros A o AAAA, estás autorizado a intentar una conexión SMTP con ellos
- Si no es ninguno de los casos anteriores pero si existe un registro de tipo CNAME (un alias a otro dominio), prueba en el otro dominio utilizando las mismas reglas

# Resolving Mail Domains

- Una de las mejores razones para realizar una búsqueda DNS a mano es para buscar el servidor de correo que le corresponde a una dirección
- Ejemplo: programa “**dns\_mx.py**”

```
jose@salmon:~$./dns_mx.py python.org
The domain 'python.org' has explicit MX records!
try the servers in this order:
Priority: 50 Hostname: mail.python.org
 Hostname mail.python.org = A 82.94.164.162
```

```
jose@salmon:~$./dns_mx.py uca.es
The domain 'uca.es' has explicit MX records!
try the servers in this order:
Priority: 10 Hostname: merlin.uca.es
 Hostname merlin.uca.es = A 150.214.86.11
Priority: 12 Hostname: saruman.uca.es
 Hostname saruman.uca.es = A 150.214.86.11
Priority: 15 Hostname: horus.uca.es
 Hostname horus.uca.es = A 150.214.86.11
Priority: 20 Hostname: smtp1.uca.es
 Hostname smtp1.uca.es = A 150.214.86.11
Priority: 40 Hostname: melchor.uca.es
 Hostname melchor.uca.es = A 150.214.86.11
```

# Actividad 2: whois

Implementa un programa de consola que responda a peticiones whois por parte de un usuario utilizando la biblioteca whois. Instálala con **sudo pip install whois** y también instala el programa utilizado por esta biblioteca (**sudo apt-get install whois**).

Nota: no llames whois.py al archivo o dará problemas al importar la biblioteca whois

Ejemplo:

Programa Whois. Escribe 's' para salir.

Dirección:google.com

\*\*\*\*\*

Nombre de dominio: google.com

Registro: MARKMONITOR INC.

Fecha de creación: 1997-09-15 00:00:00

Última actualización: 2011-07-20 00:00:00

Caduca: 2020-09-14 00:00:00

\*\*\*\*\*

Dirección:s

Saliendo...

# Preguntas de repaso

- ¿Qué es un DNS?
- ¿Podrías acceder a una página web sin DNS?
- ¿Por qué es necesario IPv6?
- ¿Hay alguna diferencia entre un nombre de host y un nombre de dominio?
- ¿En qué situación utilizarías el método `getaddrinfo()`?
- ¿En qué situación utilizarías un DNS?

# Socket names

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- Prototype in C
- When we **create** a socket we have to provide 3 arguments:
- **Address family (domain)**. It determines the type of network: internet, Unix, Appletalk, Bluetooth, ...
  - Typically AF\_INET
- **Socket type**. It determines the transfer method we will use:
  - Even though each family can have its own types, these are usually independent of the family
  - SOCK\_DGRAM (uses packets)
  - SOCK\_STREAM (uses flow control and is reliable)
- **Protocol**.
  - There is usually only one option set as the family address and the socket type. That's why it's usually left blank (a zero) so it's chosen automatically
  - In our case: TCP and UDP

# Address Families in `/usr/include/bits/socket.h`

`/* Address families. */`

```
#define AF_UNSPEC PF_UNSPEC
#define AF_LOCAL PF_LOCAL
#define AF_UNIX PF_UNIX
#define AF_FILE PF_FILE
#define AF_INET PF_INET
#define AF_AX25 PF_AX25
#define AF_IPX PF_IPX
#define AF_APPLETALK PF_APPLETALK
#define AF_NETROM PF_NETROM
#define AF_BRIDGE PF_BRIDGE
#define AF_ATMPVC PF_ATMPVC
#define AF_X25 PF_X25
#define AF_INET6 PF_INET6
#define AF_ROSE PF_ROSE
#define AF_DECnet PF_DECnet
#define AF_NETBEUI PF_NETBEUI
#define AF_SECURITY PF_SECURITY
#define AF_KEY PF_KEY
```

```
#define AF_NETLINK PF_NETLINK
#define AF_ROUTE PF_ROUTE
#define AF_PACKET PF_PACKET
#define AF_ASH PF_ASH
#define AF_ECONET PF_ECONET
#define AF_ATMSVC PF_ATMSVC
#define AF_RDS PF_RDS
#define AF_SNA PF_SNA
#define AF_IRDA PF_IRDA
#define AF_PPPOX PF_PPPOX
#define AF_WANPIPE PF_WANPIPE
#define AF_LLC PF_LLC
#define AF_CAN PF_CAN
#define AF_TIPC PF_TIPC
#define AF_BLUETOOTH PF_BLUETOOTH
#define AF_IUCV PF_IUCV
#define AF_RXRPC PF_RXRPC
#define AF_ISDN PF_ISDN
#define AF_PHONET PF_PHONET
#define AF_IEEE802154 PF_IEEE802154
#define AF_MAX PF_MAX
```

# Socket types in `/usr/include/bits/socket.h`

```
/* Types of sockets. */
enum __socket_type
{
 SOCK_STREAM = 1, /* Sequenced, reliable, connection-based byte streams. */
#define SOCK_STREAM SOCK_STREAM
 SOCK_DGRAM = 2, /* Connectionless, unreliable datagrams of fixed maximum length. */
#define SOCK_DGRAM SOCK_DGRAM
 SOCK_RAW = 3, /* Raw protocol interface. */
#define SOCK_RAW SOCK_RAW
 SOCK_RDM = 4, /* Reliably-delivered messages. */
#define SOCK_RDM SOCK_RDM
 SOCK_SEQPACKET = 5, /* Sequenced, reliable, connection-based, datagrams of
 fixed maximum length. */
#define SOCK_SEQPACKET SOCK_SEQPACKET
 SOCK_DCCP = 6, /* Datagram Congestion Control Protocol. */
#define SOCK_DCCP SOCK_DCCP
 SOCK_PACKET = 10, /* Linux specific way of getting packets at the dev level. For writing
 rarp and other similar things on the user level. */
#define SOCK_PACKET SOCK_PACKET

 /* Flags to be ORed into the type parameter of socket and socketpair and
 used for the flags parameter of paccept. */

 SOCK_CLOEXEC = 02000000, /* Atomically set close-on-exec flag for the new descriptor(s). */
#define SOCK_CLOEXEC SOCK_CLOEXEC
 SOCK_NONBLOCK = 04000 /* Atomically mark descriptor(s) as non-blocking. */
#define SOCK_NONBLOCK SOCK_NONBLOCK
};
```



# Address Families and socket types in Python:

```
import socket; dir(socket)
```

```
'AF_APPLETALK',
'AF_ASH',
'AF_ATMPVC',
'AF_ATMSVC',
'AF_AX25',
'AF_BLUETOOTH',
'AF_BRIDGE',
'AF_DECnet',
'AF_ECONET',
'AF_INET',
'AF_INET6',
'AF_IPX',
'AF_IRDA',
'AF_KEY',
```

```
'AF_LLC',
'AF_NETBEUI',
'AF_NETLINK',
'AF_NETROM',
'AF_PACKET',
'AF_PPPOX',
'AF_ROSE',
'AF_ROUTE',
'AF_SECURITY',
'AF_SNA',
'AF_TIPC',
'AF_UNIX',
'AF_UNSPEC',
'AF_WANPIPE',
'AF_X25'
```

```
'SOCK_DGRAM',
'SOCK_RAW',
'SOCK_RDM',
'SOCK_SEQPACKET',
'SOCK_STREAM'
```

# Python: protocols for the family AF\_INET

'IPPROTO\_AH',  
'IPPROTO\_DSTOPTS',  
'IPPROTO\_EGP',  
'IPPROTO\_ESP',  
'IPPROTO\_FRAGMENT',  
'IPPROTO\_GRE',  
'IPPROTO\_HOPOPTS',  
'IPPROTO\_ICMP',  
'IPPROTO\_ICMPV6',  
'IPPROTO\_IDP',  
'IPPROTO\_IGMP',

'IPPROTO\_IP',  
'IPPROTO\_IPIP',  
'IPPROTO\_IPV6',  
'IPPROTO\_NONE',  
'IPPROTO\_PIM',  
'IPPROTO\_PUP',  
'IPPROTO\_RAW',  
'IPPROTO\_ROUTING',  
'IPPROTO\_RSVP',  
**'IPPROTO\_TCP'**,  
'IPPROTO\_TP',  
**'IPPROTO\_UDP'**

# Protocols for the family AF\_INET

in the file: /usr/include/netinet/in.h

```
/* Standard well-defined IP protocols. */
enum
{
 IPPROTO_IP = 0, /* Dummy protocol for TCP. */
#define IPPROTO_IP IPPROTO_IP

 IPPROTO_IPIP = 4, /* IPIP tunnels (older KA9Q tunnels use 94) */
#define IPPROTO_IPIP IPPROTO_IPIP
 IPPROTO_TCP = 6, /* Transmission Control Protocol. */
#define IPPROTO_TCP IPPROTO_TCP
 IPPROTO_EGP = 8, /* Exterior Gateway Protocol. */
#define IPPROTO_EGP IPPROTO_EGP
 IPPROTO_PUP = 12, /* PUP protocol. */
#define IPPROTO_PUP IPPROTO_PUP
 IPPROTO_UDP = 17, /* User Datagram Protocol. */
#define IPPROTO_UDP IPPROTO_UDP

 IPPROTO_IPV6 = 41, /* IPv6 header. */
#define IPPROTO_IPV6 IPPROTO_IPV6

};
```

# Address

- Once the **family**, **type** and **protocol** are set
- When we want to connect a socket we have to provide more parameters
- In the case of **IPv4**, 2 additional parameters:
  - **Host**: 32 bit integer =  $4 \times 8$  bits
  - **Port**: 16 bit integer
- In the case of **IPv6**, 4 additional parameters:
  - **Host**: 128 bit integer =  $8 \times 16$  bits
  - **Port**: 16 bit integer
  - **Flowinfo** unicast, anycast, multicast
  - **Scopeid**. In which part of the network the address is valid: link, global, etc
- In other cases the number of extra parameters may vary

# IP address and ARP

- The IP address is used by the IP protocol for routing
- Although for computers in the same network, communication is done using physical addresses (MAC)
- There is a service **ARP** (Address Resolution Protocol) and **RARP** that maps IP addresses to physical **MAC** addresses (Media Access Control) and viceversa
- ARP manages a caché of physical addresses in the network
- In Unix we have the commands “arp” and “rarp” to handle this caché.
- If a computer must connected to the network that is not in the caché, a broadcast is sent called **ARP request frame**.
- “**arp**” is used when the IP address is known but not the MAC
- “**rarp**” (reverse), when the MAC is known but not the IP address.
- However, we can only work with the IP (transport).

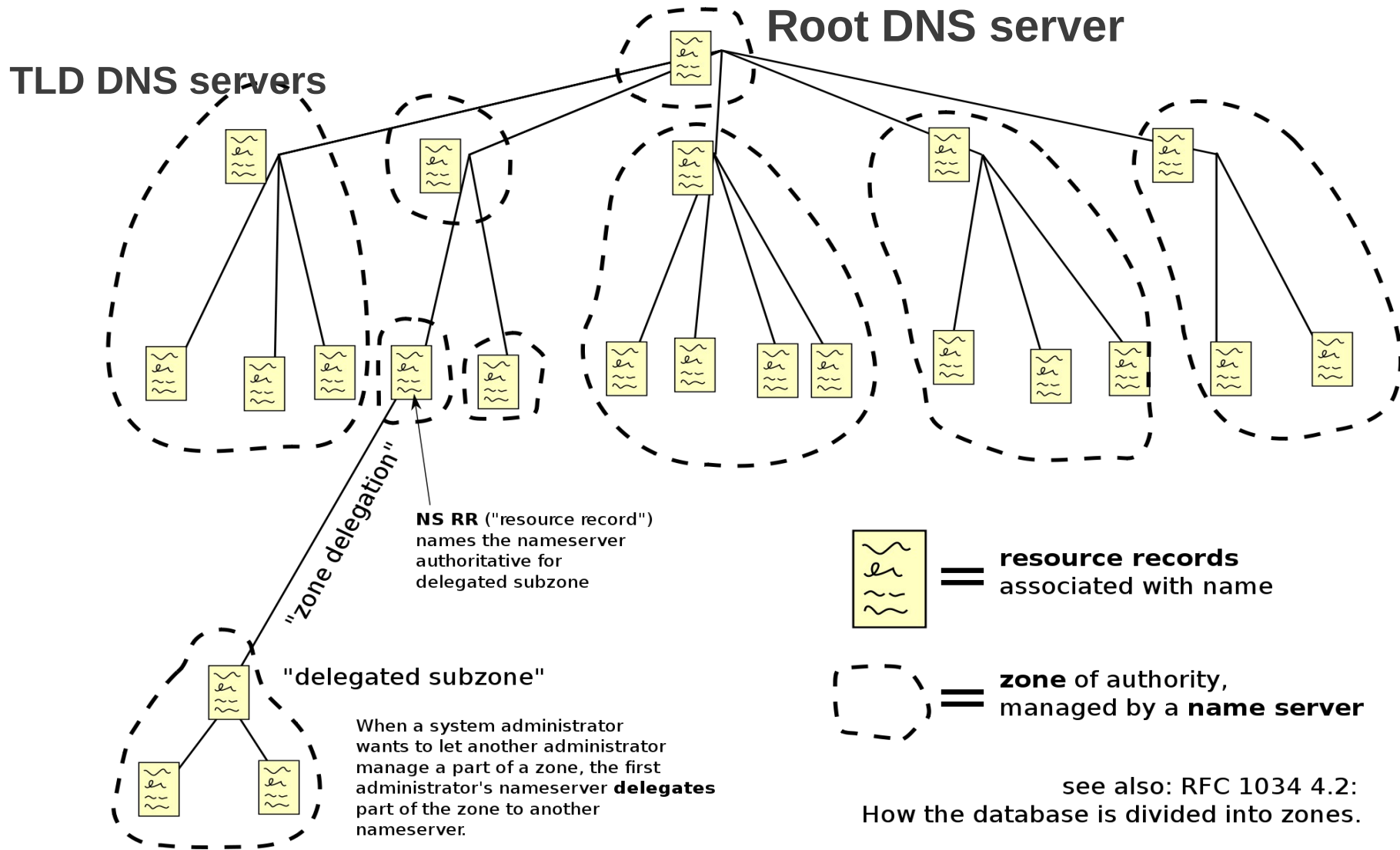
# IP Address

- Very often what we know is the name of the host, for example “[www.uca.es](http://www.uca.es)”
- To look up the IP address of a name we need a name service or DNS (Domain Name System)
- DNS is a distributed hierarchical database that stores information associated to domain names
- DNS associates different types of information to each name
- The most frequent uses are the assignment of domain names to IP addresses and the localization of email servers on each domain.
- DNS has three main components:
  - **DNS Clients:** they send DNS requests for domain resolution to a DNS server
  - **DNS servers:** Recursive servers are capable of forwarding the request to a different server if they don't have the requested address
  - And **authority zones**, parts of the domain name space that store the data
    - Each authority zone has at least a domain name and possibly subdomains

# DNS

- **TLD (Top Level Domains):** net, org, gov, uk, es ...possible suffixes for valid domain names
  - Each TLD has its own servers and organization that manages them
- **Domain name:** Domain names that organisations add to their hosts like python.org, uca.es, etc.
  - You must pay every year but it allows you to have as many hosts as you want on the same domain
- **Fully qualified domain name (FQDN):** It's usually formed by the computer name and the domain name
- **Hostname:**
  - It's an ambiguous term. Sometimes it refers to FQDN and other times to the short computer name without the domain.
  - FQDN identifies a computer from anywhere
  - The short name only allows access within the organization

# Domain Name Space





# Resource Record of DNS Zone File

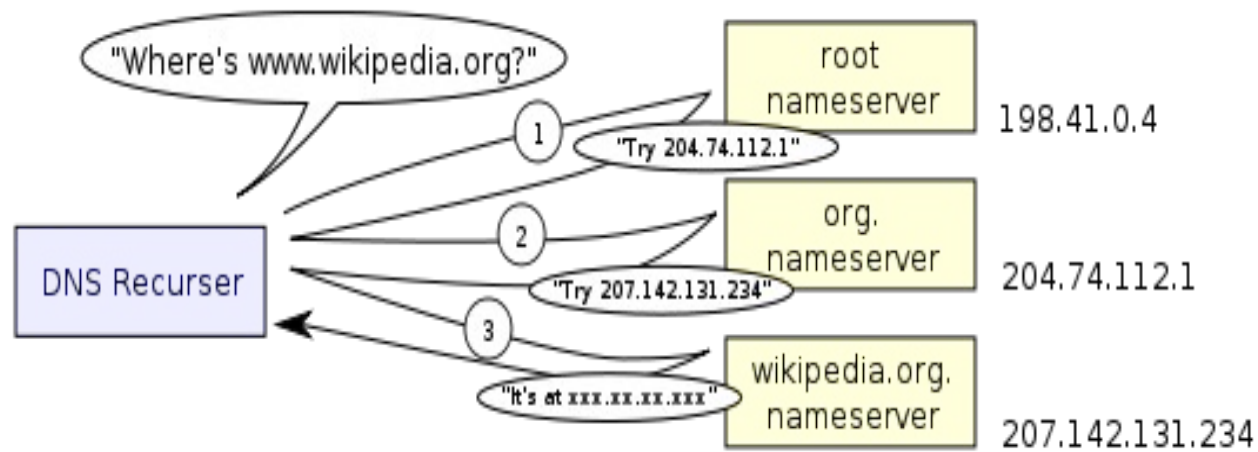
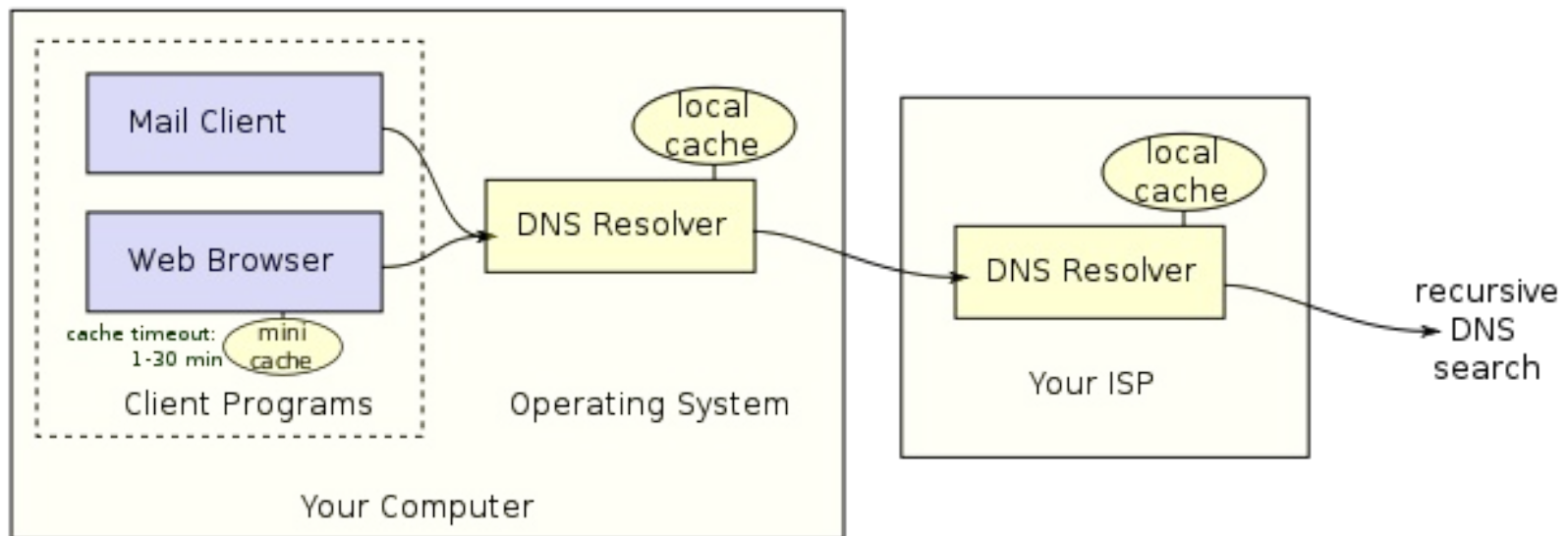
| Description |                                                                                           | Length (octets) |
|-------------|-------------------------------------------------------------------------------------------|-----------------|
| NAME (fqdn) | Name of the node to which this record pertains                                            | (variable)      |
| TYPE        | Type of RR in numeric form (e.g. 15 for MX RRs)                                           | 2               |
| CLASS       | Class code                                                                                | 2               |
| TTL         | Count of seconds that the RR stays valid (The maximum is 231-1, which is about 68 years.) | 4               |
| RDLLENGTH   | Length of RDATA field                                                                     | 2               |
| RDATA       | Additional RR-specific data                                                               | (variable)      |

# Resource records Type(some)

| Type  | Description           | Function                                                                                             |
|-------|-----------------------|------------------------------------------------------------------------------------------------------|
| A     | address record        | Returns a 32-bit IPv4 address, most commonly used to map hostnames to an IP address of the host      |
| AAAA  | IPv6 address record   | Returns a 128-bit IPv6 address, most commonly used to map hostnames to an IP address of the host.    |
| CERT  | Certificate record    | Stores PKIX, SPKI, PGP, etc.                                                                         |
| CNAME | Canonical name record | Alias of one name to another: the DNS lookup will continue by retrying the lookup with the new name. |
| DNAME | delegation name       | DNAME creates an alias for a name and all its subnames                                               |
| LOC   | Location record       | Specifies a geographical location associated with a domain name                                      |
| MX    | mail exchange record  | Maps a domain name to a list of message transfer agents for that domain                              |
| NS    | name server record    | Delegates a DNS zone to use the given authoritative name servers                                     |
| PTR   | pointer record        | Pointer to a canonical name. The most common use is for implementing reverse DNS lookups             |

# To look up an address given a name, the OS

- It's looked up locally in the files **/etc/hosts** or **/windows/system32/drivers/hosts**
- It also has a caché of previous searches:
- If it doesn't find it (or the entry in the caché has expired), then it sends messages using the **dns protocol** to the servers that appear in the file **/etc/resolv.conf** that are usually name servers in our domain
- Obviously, this changes from one operating system to another and configuration
- The name servers in our domain, look up in their database (of their domain) and in their own caché (in case the address was used before)
- Each entry in the caché has an expiry date
- If it doesn't find it, it starts a hierarchical search on the TLD servers (the address of these servers are permanent)
- The search is forwarded to the domain servers of the subdomains
- This can take a long time



**From  
Wikipedia**

# Search examples

```
jose@salmon:~$ nslookup www.uca.es
```

```
Server: 10.141.1.3
```

```
Address: 10.141.1.3#53
```

Véanse los números de puerto predefinidos:

```
Name: www.uca.es
```

```
Address: 150.214.86.11
```

[http://en.wikipedia.org/wiki/List\\_of\\_TCP\\_and\\_UDP\\_port\\_numbers](http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers)

Archivos: `/etc/resolv.conf` y `/etc/hosts` en ubuntu

```
Generated by NetworkManager
```

```
domain uca.es
```

```
search uca.es
```

```
nameserver 10.141.1.3
```

```
nameserver 150.214.76.3
```

```
nameserver 150.214.76.26
```

```
127.0.0.1 localhost
```

```
127.0.1.1 salmon
```

```
The following lines are desirable for IPv6 capable hosts
```

```
::1 localhost ip6-localhost ip6-loopback
```

```
fe00::0 ip6-localnet
```

```
ff00::0 ip6-mcastprefix
```

```
ff02::1 ip6-allnodes
```

```
ff02::2 ip6-allrouters
```

```
ff02::3 ip6-allhosts
```

## Archivo: /windows/system32/drivers/etc/hosts de windows XP

```
Copyright (c) 1993-1999 Microsoft Corp.
#
Éste es un ejemplo de archivo HOSTS usado por Microsoft TCP/IP para Windows.
#
Este archivo contiene las asignaciones de las direcciones IP a los nombres de
host. Cada entrada debe permanecer en una línea individual. La dirección IP
debe ponerse en la primera columna, seguida del nombre de host correspondiente.
La dirección IP y el nombre de host deben separarse con al menos un espacio.
#
#
Por ejemplo:
#
102.54.94.97 rhino.acme.com # servidor origen
38.25.63.10 x.acme.com # host cliente x

127.0.0.1 localhost

127.0.0.1 activate.adobe.com
127.0.0.1 practivate.adobe.com
127.0.0.1 ereg.adobe.com
127.0.0.1 activate.wip3.adobe.com
127.0.0.1 wip3.adobe.com
127.0.0.1 3dns-3.adobe.com
127.0.0.1 3dns-2.adobe.com
127.0.0.1 adobe-dns.adobe.com
127.0.0.1 adobe-dns-2.adobe.com
127.0.0.1 adobe-dns-3.adobe.com
127.0.0.1 ereg.wip3.adobe.com
127.0.0.1 activate-sea.adobe.com
127.0.0.1 wwis-dubc1-vip60.adobe.com
127.0.0.1 activate-sjc0.adobe.com
```

Domain Information Gopher  
To look up domain information

```
jose@salmon:~$ dig flikr.com ANY
```

```
; <<>> DiG 9.7.0-P1 <<>> flikr.com ANY
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 29029
;; flags: qr rd ra; QUERY: 1, ANSWER: 7, AUTHORITY: 5, ADDITIONAL: 0

;; QUESTION SECTION:
;flikr.com. IN ANY

;; ANSWER SECTION:
flikr.com. 20943 IN A 68.180.206.184
flikr.com. 20943 IN A 206.190.60.37
flikr.com. 172143 IN NS ns4.yahoo.com.
flikr.com. 172143 IN NS ns5.yahoo.com.
flikr.com. 172143 IN NS ns1.yahoo.com.
flikr.com. 172143 IN NS ns2.yahoo.com.
flikr.com. 172143 IN NS ns3.yahoo.com.

;; AUTHORITY SECTION:
flikr.com. 172143 IN NS ns3.yahoo.com.
flikr.com. 172143 IN NS ns4.yahoo.com.
flikr.com. 172143 IN NS ns5.yahoo.com.
flikr.com. 172143 IN NS ns1.yahoo.com.
flikr.com. 172143 IN NS ns2.yahoo.com.

;; Query time: 1 msec
;; SERVER: 10.141.1.3#53(10.141.1.3)
;; WHEN: Thu Mar 31 10:36:07 2011
;; MSG SIZE rcvd: 225
```

```
jose@salmon:~jose$ dig -x 10.141.9.175

; <<>> DiG 9.7.0-P1 <<>> -x 10.141.9.175
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 58215
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1,
ADDITIONAL: 1

;; QUESTION SECTION:
;175.9.141.10.in-addr.arpa. IN PTR

;; ANSWER SECTION:
175.9.141.10.in-addr.arpa. 172800 INPTR salmon2.uca.es.

;; AUTHORITY SECTION:
141.10.in-addr.arpa. 172800 IN NS baltasar.uca.es.

;; ADDITIONAL SECTION:
baltasar.uca.es.172800 IN A 150.214.76.3

;; Query time: 0 msec
;; SERVER: 10.141.1.3#53(10.141.1.3)
;; WHEN: Thu Mar 31 12:07:13 2011
;; MSG SIZE rcvd: 110
```



```
jose@salmon:~$ whois -d -B -h whois.ripe.net 150.214.86.11
% This is the RIPE Database query service.
% The objects are in RPSL format.
%
% Information related to '150.214.0.0 - 150.214.255.255'

inetnum: 150.214.0.0 - 150.214.255.255
netname: CICA
descr: Centro Informatico Cientifico de Andalucia
descr: Sevilla
country: ES
. . . .
% Information related to '150.214.0.0/16AS766'
route: 150.214.0.0/16
descr: RICA
origin: AS766
mnt-by: REDIRIS-NMC
. . . .
% Information related to '214.150.in-addr.arpa'
domain: 214.150.in-addr.arpa
descr: Centro Informatico Cientifico de Andalucia
descr: Av. de Reina Mercedes SN
descr: Sevilla, 41012
admin-c: SBG9-RIPE
tech-c: SBG9-RIPE
zone-c: SBG9-RIPE
nserver: dns1.cica.es
nserver: dns2.cica.es
nserver: sun.rediris.es
nserver: chico.rediris.es
nserver: ns.ripe.net
. . . .
```

# It's better to use the OS services

- As we can see the structure of an address depends on the family: Ipv4, Ipv6, AppleTalk, ...
- Moreover, when we look up an address it is convenient to use the OS resources (first the host and cache file, then the name servers in our domain. Initiate a hierarchical search only when it's necessary)
- **Python** provides the function: **`socket.getaddrinfo()`** that uses the operating system, it's able to handle many address families and can be used in many different ways
- In any case, sometimes we want to perform a search on the DNS servers ourselves

```
socket.getaddrinfo(host, port, family=0,
 socktype=0, proto=0, flags=0)
```

- The best way to handle all these parameters is using the function **getaddrinfo()**
- Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for **creating a socket** connected to that service.
- **host** is a domain name, a string representation of an IPv4/v6 address or None.
- **port** is a string service name such as 'http', a numeric port number or None.
- By passing None as the value of host and port, you can pass NULL to the **underlying C API**.
- The **family**, **socktype** and **proto** arguments can be **optionally** specified in order to narrow the list of addresses returned.
- Passing zero as a value for each of these arguments selects the full range of results.
- The **flags** argument can be one or several of the **AI\_\*** constants, and will influence how results are computed and returned.
  - For example, **AI\_NUMERICHOST** will disable domain name resolution and will raise an error if host is a domain name.

```
socket.getaddrinfo(host, port, family=0,
 socktype=0, proto=0, flags=0)
```

- The function returns a list of 5-tuples with the following structure:
  - **(family, socktype, protocol, canonname, sockaddr)**
- In these tuples, family, socktype, proto are all integers and **are meant to be passed** to the **socket()** function.
- canonname will be a string representing the canonical name of the host if `AI_CANONNAME` is part of the flags argument; else canonname will be empty.
- sockaddr is a tuple describing a socket address, whose format depends on the returned family:
  - (address, port) 2-tuple for `AF_INET`
  - (address, port, flow info, scope id) 4-tuple for `AF_INET6`
  - and **is meant to be passed** to the **socket.connect()** method.
- The following example fetches address information for a hypothetical TCP connection to `www.python.org` on port 80 (results may differ on your system if IPv6 isn't enabled):

```
>>> socket.getaddrinfo("www.python.org", 80, 0, 0,
 socket.SOCK_TCP)
[(2, 1, 6, '', ('82.94.164.162', 80)),
 (10, 1, 6, '', ('2001:888:2000:d::a2', 80, 0, 0))]
```

# Example: creating a socket and connection

The next example uses `socket.getaddrinfo()` to create a socket and connect it to the HTTP port at “gatech.edu”.

```
>>> import socket
>>> infolist = socket.getaddrinfo('gatech.edu', 'www')
>>> infolist
[(2, 1, 6, '', ('130.207.244.244', 80)),
 (2, 2, 17, '', ('130.207.244.244', 80))]
```

The reply shows that there are two ways to create the socket:

- 1) Type `SOCK_STREAM=1` and protocol `IPPROTO_TCP=6`
- 2) Type `SOCK_DGRAM=2` and protocol `IPPROTO_UDP=17`  
(officially HTTP supports TCP and UDP)

```
>>> ftpca = infolist[0]
>>> ftpca # Family Type Protocol Canonical-name Address
(2, 1, 6, '', ('130.207.244.244', 80))
>>> ftpca[0:3]
(2, 1, 6)
>>> s = socket.socket(*ftpca[0:3])
>>> ftpca[4]
('130.207.244.244', 80)
>>> s.connect(ftpca[4])
```

# More examples of `getaddrinfo()`: on the server side:

- We want to create a listening socket that serves “smtp” traffic using “tcp:”

```
>>> i = socket.getaddrinfo(None, 'smtp', 0, socket.SOCK_STREAM, 0, socket.AI_PASSIVE)
>>> i
[(2, 1, 6, '', ('0.0.0.0', 25)), (10, 1, 6, '', (':::', 25, 0, 0))]
>>> ftpca = i[0] # Family Type Protocol Canonical-name Address
>>> ftpca
(2, 1, 6, '', ('0.0.0.0', 25))
>>> FamTypProto = ftpca[0:3]
>>> FamTypProto
(2, 1, 6)
>>> s = socket.socket(*FamTypProto)
```

- Now we want to create a listening socket that serves “dns” traffic using “udp”:

```
>>> j = socket.getaddrinfo(None, 53, 0, socket.SOCK_DGRAM, 0, socket.AI_PASSIVE)
>>> j
[(2, 2, 17, '', ('0.0.0.0', 53)), (10, 2, 17, '', (':::', 53, 0, 0))]
>>> s2 = socket.socket(*j[0][0:3])
```

# Extract from <http://www.iana.org/assignments/port-numbers> and “/etc/services”

## PORT NUMBERS

(last updated 2011-03-29)

The port numbers are divided into three ranges: the Well Known Ports, the Registered Ports, and the Dynamic and/or Private Ports.

The Well Known Ports are those from 0 through 1023.

. . . . .

Port Assignments:

| Keyword       | Decimal       | Description                    | References |
|---------------|---------------|--------------------------------|------------|
| -----         | -----         | -----                          | -----      |
|               | 0/tcp         | Reserved                       |            |
|               | 0/udp         | Reserved                       |            |
| #             |               | Jon Postel <postel&isi.edu>    |            |
| spr-itunes    | 0/tcp         | Shirt Pocket netTunes          |            |
| spl-itunes    | 0/tcp         | Shirt Pocket launchTunes       |            |
| . . . . .     |               |                                |            |
| #             |               | Rick Adams <rick&UUNET.UU.NET> |            |
| <b>smtp</b>   | <b>25/tcp</b> | <b>Simple Mail Transfer</b>    |            |
| <b>smtp</b>   | <b>25/udp</b> | <b>Simple Mail Transfer</b>    |            |
| . . . . .     |               |                                |            |
| <b>domain</b> | <b>53/tcp</b> | <b>Domain Name Server</b>      |            |
| <b>domain</b> | <b>53/udp</b> | <b>Domain Name Server</b>      |            |
| . . . . .     |               |                                |            |

## More examples of `getaddrinfo()`: client and server

- Now we assume that a client wants to connect with certain known server: [www.uca.es](http://www.uca.es) using the HTTP port and a given family (INET) and type (SOCK\_STREAM).
- Or a server wants to connect to “localhost” using the port 1060, the family INET6 and the protocol “udp”:

```
>>> i = socket.getaddrinfo('www.uca.es', 'http', socket.AF_INET, socket.SOCK_STREAM)
>>> i
[(2, 1, 6, '', ('150.214.86.11', 80))]
>>> s = socket.socket(*i[0][0:3])
>>> s.connect(i[0][4])

>>> j = socket.getaddrinfo('localhost', 1060, socket.AF_INET6, 0, socket.IPPROTO_TCP)
>>> j
[(10, 1, 6, '', ('::1', 1060, 0, 0))]
>>> ftpca = j[0] # Family Type Protocol Canonical-name Address
>>> k = socket.socket(*ftpca[0:3])
>>> a = j[0][4] # Address
>>> a
('::1', 1060, 0, 0)
>>> k.bind(a)
```



# Exercise `getaddrinfo()`

In what ways can you connect to “[www.uca.es](http://www.uca.es)” ?

## More examples of getaddrinfo(): flags

```
>>> i = socket.getaddrinfo('iana.org', 'www', 0, socket.SOCK_STREAM, 0,
 socket.AI_ADDRCONFIG | socket.AI_V4MAPPED)
>>> i
[(2, 1, 6, '', ('192.0.43.8', 80))]
```

```
>>> i = socket.getaddrinfo('iana.org', 'www', 0, socket.SOCK_STREAM, 0,
 socket.AI_ADDRCONFIG | socket.AI_V4MAPPED | socket.AI_CANONNAME)
>>> i
[(2, 1, 6, '43-8.any.icann.org', ('192.0.43.8', 80))]
```

**AI\_ADDRCONFIG** filters the addresses that aren't reachable from your computer

**AI\_V4MAPPED** returns the IPv4 addresses that you can use but coded as IPv6 (in case that your machine has an IPv6 interface but the service you are using only uses IPv4)

- **AI\_CANONNAME** does DNS searches and returns the canonical name of the host
- Last, **AI\_CANONNAME**. A server finds the canonical name of a client that has connected to it

```
>>> sc, (direcc, puerto) = s.accept()
>>> i = socket.getaddrinfo(direcc, puerto, sc.family, sc.type,
 sc.proto, socket.AI_CANONNAME)
>>> print "i =", i
i = [(2, 1, 6, 'rr.pmta.wikimedia.org', ('208.80.152.2', 80))]
```

# Example use in a program

- The program “**www\_ping.py**” uses the information returned by **getaddrinfo()** to check if a machine can connect using tcp
- Something similar to “ping”
- The use of **getaddrinfo()** allows us to write a generic program, independent of the family: Ipv4, Ipv6, AppleTalk, etc. y del protocolo

```
jose@salmon:~$./www_ping.py mit.edu
Éxito: host WEB.MIT.EDU está escuchando por el puerto 80
jose@salmon:~$./www_ping.py smtp.google.com
fallo de la red: Connection timed out
jose@salmon:~$./www_ping.py ordenador-inexistente.com
Fallo del servicio de nombres: No address associated with hostname
```

# One more example

- The program “**forward\_reverse.py**”
- Looks up the IP address of a name, and viceversa, looks up the canonical name of that address
- Last, compare both names

```
jose@salmon:~$./forward_reverse.py flickr.com
flickr.com has IP address 68.142.214.24
68.142.214.24 has the canonical name www.flickr.vip.mud.yahoo.com
WARNING! The forward and reversenames belongs to a different organization
jose@salmon:~$./forward_reverse.py mit.edu
mit.edu has IP address 18.9.22.69
18.9.22.69 has the canonical name WEB.MIT.EDU
The forward and reverse names have a lot in common
jose@salmon:~$./forward_reverse.py www.uca.es
www.uca.es has IP address 150.214.86.11
150.214.86.11 has the canonical name www.uca.es
Estupendo, los nombres coinciden completamente
```

# Packet pydns

- `pip install pydns`
- To do DNS lookups manually
- Use example: `dns_basic.py`

```
jose@salmon:~$./dns_basic.py python.org
python.org IN A '82.94.164.162'
python.org IN AAAA ' \x01\x08\x88 \x00\x00\r\x00\x00\x00\x00\x00\x00\xa2'
python.org IN MX (50, 'mail.python.org')
python.org IN NS 'ns2.xs4all.nl'
python.org IN NS 'ns.xs4all.nl'
jose@salmon:~$./dns_basic.py uca.es
uca.es IN A '150.214.86.11'
uca.es IN MX (40, 'melchor.uca.es')
uca.es IN MX (10, 'merlin.uca.es')
uca.es IN MX (12, 'saruman.uca.es')
uca.es IN MX (15, 'horus.uca.es')
uca.es IN MX (20, 'smtp1.uca.es')
uca.es IN NS 'dns1.cica.es'
uca.es IN NS 'dns2.cica.es'
uca.es IN NS 'nimue.uca.es'
uca.es IN NS 'baltasar.uca.es'
```

# RFC 5321

- This document specifies the rules to resolve email domains:
- If there are registers of type MX, you must contact these SMTP servers and return an error to the user if none of them accepted this message
- If there are no MX registers but there are A registers or AAAA, then you are authorized to try an SMTP connection to them
- If none of the above but there is a register of type CNAME (an alias of a different domain), try the other domain using the same rules
-

# Resolving Mail Domains

- One of the best reasons to do a DNS search manually is to search the email server that corresponds to an address
- Example: program “**dns\_mx.py**”

```
jose@salmon:~$./dns_mx.py python.org
The domain 'python.org' has explicit MX records!
try the servers in this order:
Priority: 50 Hostname: mail.python.org
 Hostname mail.python.org = A 82.94.164.162
```

```
jose@salmon:~$./dns_mx.py uca.es
The domain 'uca.es' has explicit MX records!
try the servers in this order:
Priority: 10 Hostname: merlin.uca.es
 Hostname merlin.uca.es = A 150.214.86.11
Priority: 12 Hostname: saruman.uca.es
 Hostname saruman.uca.es = A 150.214.86.11
Priority: 15 Hostname: horus.uca.es
 Hostname horus.uca.es = A 150.214.86.11
Priority: 20 Hostname: smtp1.uca.es
 Hostname smtp1.uca.es = A 150.214.86.11
Priority: 40 Hostname: melchor.uca.es
 Hostname melchor.uca.es = A 150.214.86.11
```

# Exercise 2: whois

Write a console program that replies to whois requests by a client using the whois library. Install it using **sudo pip install whois** and also install the program used by this library (**sudo apt-get install whois**).

Note: don't call the file whois.py or it will give problems when importing the whois library

Example

Program Whois. Write 'e' to exit.

Address:google.com

\*\*\*\*\*

Domain name: google.com

Registrar: MARKMONITOR INC.

Creation date: 1997-09-15 00:00:00

Last update: 2011-07-20 00:00:00

Expires on: 2020-09-14 00:00:00

\*\*\*\*\*

Address:e

Closing...



# Review questions

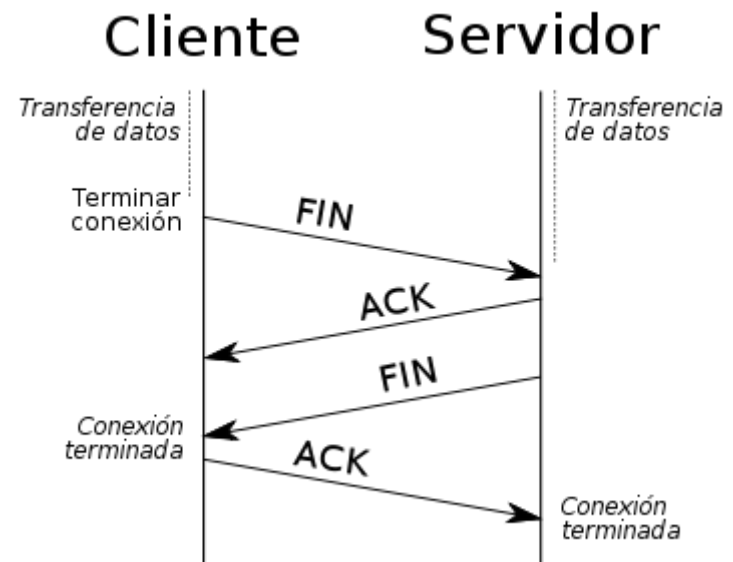
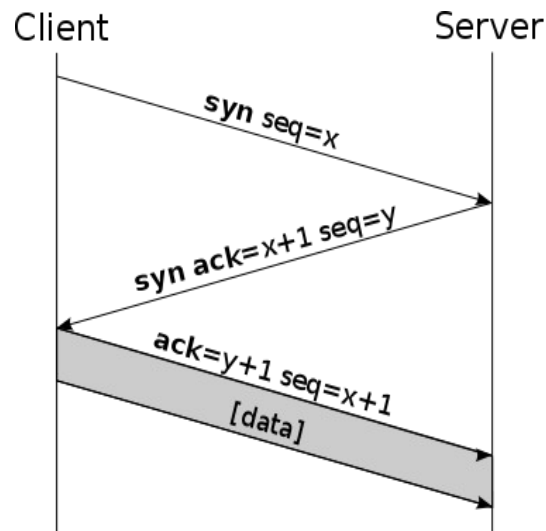
- What is DNS?
- Would you be able to access a web page without using DNS?
- Why is IPv6 necessary?
- Is there a difference between a hostname and a domain name?
- When would you use `getaddrinfo()`?
- When would you use DNS?

# Protocolo TCP

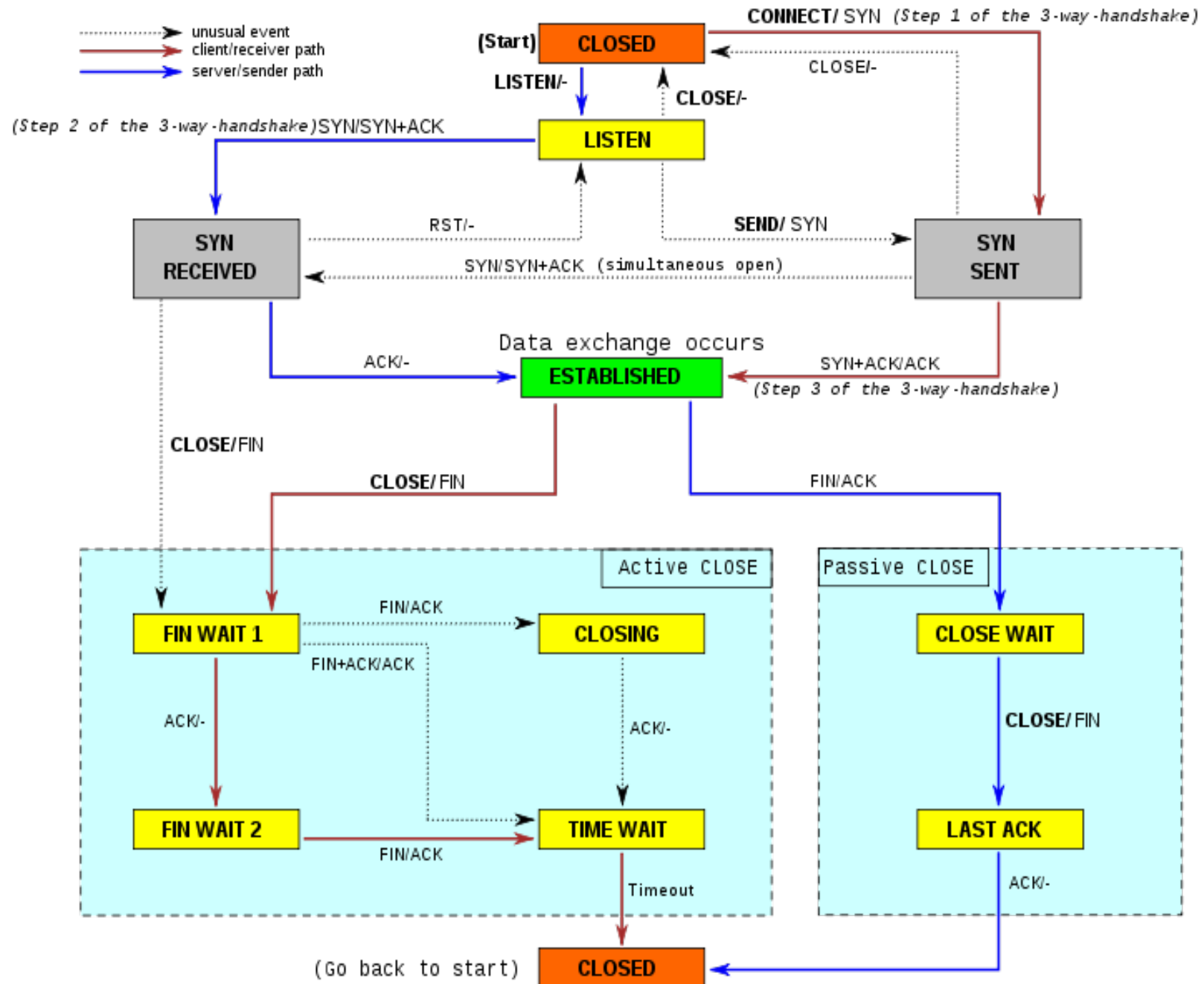
- Orientado a conexión
- Streams
- Garantiza fiabilidad (maneja automáticamente algunos de los problemas de UDP): paquetes que no llegan, que llegan duplicados, fuera de orden.
  - N° de secuencia del paquete (n° de bytes transmitidos) si  $\text{seq\_n}^{\circ} = 7200$  y  $\text{tamaño\_paquete} = 1024 \Rightarrow \text{nuevo seq\_n}^{\circ} = 8224$
  - Inicializado a random (para dificultar “Connection hijacking”).
  - Envía muchos paquetes del tirón (antes de esperar confirmación) “TCP window size” (entre 2 y 65,535 bytes \* window scaling)
  - El receptor puede regular el “window size” realizando un “control de flujo” (puede ralentizar o pausar la conexión) especificando el máximo buffer que está dispuesto a admitir.
  - También existe control de congestión de la red
- Es el protocolo habitual sobre el que corren: HTTP, SSH, etc.

# 3 way handshake

- Establecer una conexión necesita 3 paquetes: SYN (quiero establecer comm y este es nº sec. paquete), SYN-ACK (de acuerdo, aquí está el mío), ACK-OK.
- Y otros 3 paquetes para terminar la conexión: FIN, FIN-ACK, ACK.
- O bien una cadena más larga de FIN-ACK.



# Diagrama de estados



# TCP *versus* UDP

- TCP:
  - soluciona algunos problemas de UDP (fiabilidad, congestión tráfico, ...)
  - es más lento que UDP debido al “handshake”
  - no es apropiado para transferencias en tiempo real porque pide reenviar paquetes perdidos
- UDP:
  - es más rápido (no necesita tanto “overhead”)
  - es más complicado de manejar en la lógica
  - puedes gestionar tu propio protocolo

# TCP es orientado a conexión

- **connect ( )**
  - En el “client side”
  - Dispara el protocolo de “handshake” y puede fallar
  - Establece una conexión permanente entre 2 sockets (circuito virtual)
- Existen 2 tipos de sockets en el **lado del servidor**:
- **socket pasivo (listening socket)**: por el que se desea escuchar
  - sólo hay uno en el servidor
  - tiene un (direcc\_IP\_escucha, port\_escucha)
- **socket activo ó conectado**: conectado a otro socket externo cliente con su propio (direcc\_IP\_cliente, port\_cliente)
  - Pueden existir muchos en el servidor (miles)
  - Cada uno tiene una cuadrupla diferente:  
(direcc\_IP\_escucha, port\_escucha) <=> (direcc\_IP\_cliente, port\_cliente)
  - La (direcc\_IP\_escucha, port\_escucha) es la misma para todos los sockets activos
  - Pueden funcionar como un archivo para el S.O.

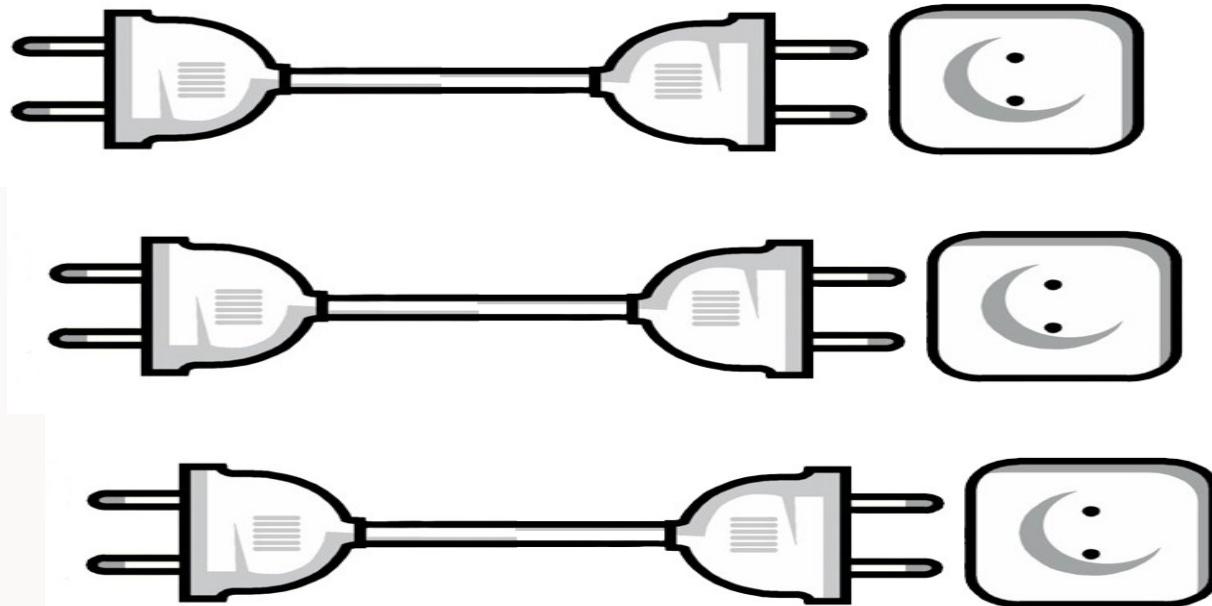
# Algoritmo para manejar un socket TCP en el servidor:

- Crear un **socket pasivo** :
  - `s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
  - `s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`
- Vamos a escuchar a ciertos hosts por cierto puerto:
  - `s.bind((HOST, PORT))`
  - `s.listen(nº máximo conexiones simultáneas)`
- Cada conexión simultánea se podría tratar en un “hilo” o “proceso” diferente (lo trataremos en otro tema)
- Bucle para siempre:
  - Crear un **socket activo** conectado: `sc, sockname = s.accept()`
  - Bucle de datos:
    - Leer datos: `datos = sc.recv(longitud_maxima)`
    - Enviar respuesta: `sc.sendall(respuesta)`
- Cerrar conexión (handshake de fin): `sc.close()`

# Socket pasivo y activos



Socket “**S**” pasivo  
Creado por  
**socket ()** con,  
hasta, tres  
conexiones  
simultáneas



Tres sockets activos “**sc**” diferentes  
creados por **accept ()** : cada uno  
se caracteriza por una cuádrupla  
distinta



# ¿Cuándo se decide?

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

**s.bind()**

bind() puede usarse tanto en el servidor como en el cliente (aunque no es lo habitual) para pedir un PORT específico

**Servidor**

**cliente**

**s.listen()**

**s.connect()**

Sólo se utiliza en el servidor y convierte al socket "s" en pasivo

En el cliente.  
Convierte al socket "s" en activo

```
sc, sockname = s.accept()
```

En el servidor. Devuelve sockets activos "sc"

# Algoritmo para manejar un socket TCP en el cliente:

- Crear un **socket**:
  - **`s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`**
- Al socket “**s**” Se le asigna la **dirección IP** de su ordenador
- El S.O. También le asigna a “**s**” un **número de puerto** aleatorio.
- Establecer conexión con el servidor:
  - **`s.connect((direccion_servidor, port_servidor))`**
    - Establece el 3-way hadshake entre cliente y servidor
- Bucle de datos:
  - Enviar datos: **`s.sendall(respuesta)`**
  - Leer respuesta: **`datos = s.recv(longitud_max)`**
- Cerrar conexión (handshake de fin): **`s.close()`**
- Ejemplo: tcp\_sixteen.py

# Conexión a través de streams (send)

- La conexión es en forma de flujo (stream) con buffers de entrada y salida
- Esto implica que “**send()**” y “**recv()**” significan, en TCP, cosas diferentes que en UDP (donde eran operaciones atómicas)
- Existen varias posibilidades para “**send()**”
  - O bien los datos son aceptados por el otro, o bien existe espacio en el buffer de salida: en ambos casos, **send()** termina y devuelve la longitud total enviada
  - Sólo se aceptan parte de los datos: **send()** termina devolviendo el número de bytes enviados
  - Si la tarjeta de red está ocupada o bien el buffer de salida está lleno, **send()** se bloquea esperando
- De forma que hay que poner a **send()** en un bucle
- Python proporciona un método “**sendall(datos)**”

# Conexión a través de streams (recv)

- **recv(nº de bytes a leer)**
  - Si no hay datos en el buffer de entrada, se bloquea hasta que lleguen los datos
  - Si existen suficientes datos en el buffer de entrada. Termina devolviendo el número de bytes que has pedido (bytes leídos)
  - Si existen datos en el buffer de entrada, pero no los suficientes, también termina pero devuelve lo que tiene (bytes leídos)
- Todo esto obliga a meter “**recv()**” en un bucle
- Python tiene un método “**sendall()**” pero sólo proporciona el método “**recv()**”
- **no existe** un método “**recv\_all()**”
- Tenemos un problema al leer:
  - En caso de conocer de antemano la longitud del mensaje enviado, se lee dicha longitud
  - Si no se conoce la longitud, podemos leer hasta final del stream
  - También podemos enviar la longitud del mensaje como parte del mensaje (equivalente al campo “**content-length:**” de HTTP)

# Otros métodos de la clase socket:

- `s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`
  - Si no se pone, el S.O. Se queda 4 minutos en el estado “time wait” (cuando se cierra un socket)
- **`sc.getsockname()`**
  - tanto para sockets activos como pasivos. Devuelve su propia dirección
- **`sc.getpeername()`**
  - sólo para sockets activos. Devuelve dirección del partner.

# deadlock

- Consiste en que cliente/servidor se quedan esperándose mutuamente de forma indefinida
- Puede ocurrir muy fácilmente en TCP debido a la utilización de un buffer de entrada y de salida de tamaño limitado
  - Buffer de escritura: almacena hasta que se pueda enviar
  - Buffer de lectura: almacena hasta que la aplicación lo pueda procesar
- Esto no ocurre en UDP
- Ejemplo: **tcp\_deadlock.py**
- Este programa llena los buffers y entonces se dispara el sistema de “control de flujo” (que puede ralentizar o paralizar el flujo) para evitar que el cliente envíe más datos (pero el cliente no lee hasta que haya escrito todo y por eso queda bloqueado)
- Soluciones posibles:
  - **s.setblocking(False)** para inhabilitar el bloqueo de **send()** y **recv()** (finalizarán inmediatamente si no pueden leer o escribir)
  - Lectura y escritura pueden dividirse en **hilos** o **procesos** diferentes: por ejemplo, un hilo escribe y otro hilo diferente lee
  - Utilizar **select()**, **poll()** ó **epoll()** del S.O. Para conseguir (**a mano**) programas “*event driven*”
  - O bien, la mejor opción, utilizar un “**event-driven**” **framework** que maneje automáticamente la lógica.

# Sockets bidireccionales

- Los sockets son bidireccionales: el mismo socket puede utilizarse para enviar “**send()**” o recibir “**recv()**” datos
- A pesar de que no siempre se utilizan las dos direcciones, no es posible crear un socket unidireccional
- La solución está en cerrar una dirección pero no la otra:
  - **shutdown(SHUT\_WR)** , **shutdown(SHUT\_RD)** , **shutdown(SHUT\_RDWR)**
- En Python (pero no en POSIX sockets):
  - Un **socket** puede **recv()** y **send()**
  - Un **archivo** puede **read()** y **write()**
  - Pero no los dos a la vez
  - Pero existe el método “**makefile()**”, que devuelve un archivo
  - También está el método “**fileno()**”

# Preguntas de repaso

- ¿A qué capa pertenece el protocolo TCP?
- ¿Qué elementos constituirían una dirección TCP/IP?
- ¿Tiene sesión el protocolo TCP?
- ¿Por qué se inicializa aleatoriamente el número de secuencia?
- ¿Cuáles son las principales diferencias entre TCP y UDP?



## Ejercicio 1: enviar mensajes de longitud no especificada

- Escriba un programa en que el cliente envíe un mensaje al servidor, y este le responda diciendo la dirección y tamaño del mensaje recibido.
- Por supuesto, hay que utilizar sockets TCP
- Hacerlo por dos procedimientos diferentes:
  1. Leyendo con **recv()**, enviando datos con **sendall()** y señalando el fin del mensaje con **shutdown()**
  2. Obtener un archivo del socket para poder leer y escribir con **read()** y **write()**, entonces hay que pasar 1º el tamaño del mensaje y, finalmente, leer dicho mensaje

Tiempo estimado: 50 minutos

## Ejercicio 2: proxy

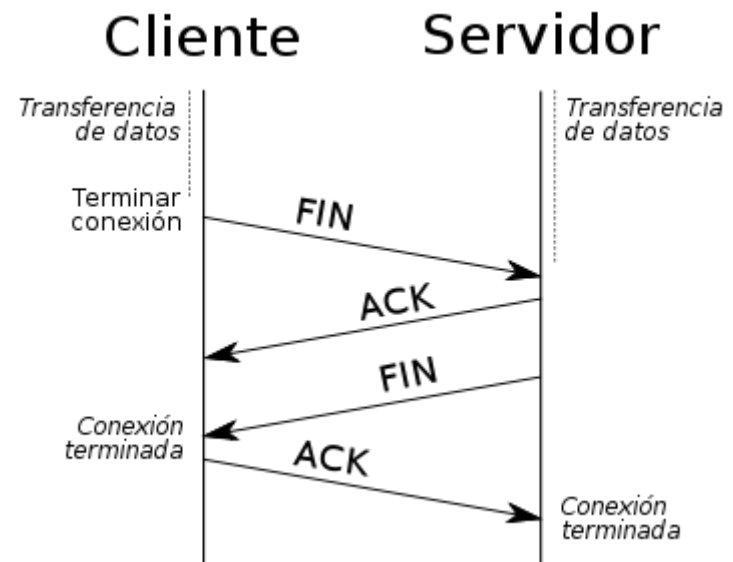
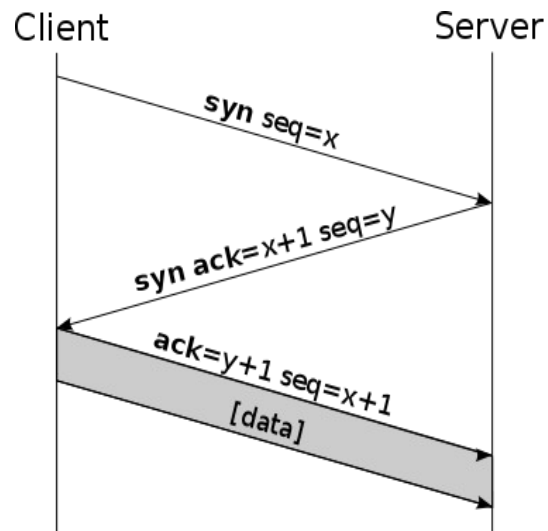
- Utilizando la segunda técnica del ejercicio anterior. Crea un servidor TCP que reciba un mensaje de un cliente y lo envíe a un segundo cliente que se ha debido conectar previamente a dicho servidor. Una vez que se envíe este mensaje al segundo cliente, se terminará la conexión.
- Tiempo estimado: 45 minutos

# TCP Protocol

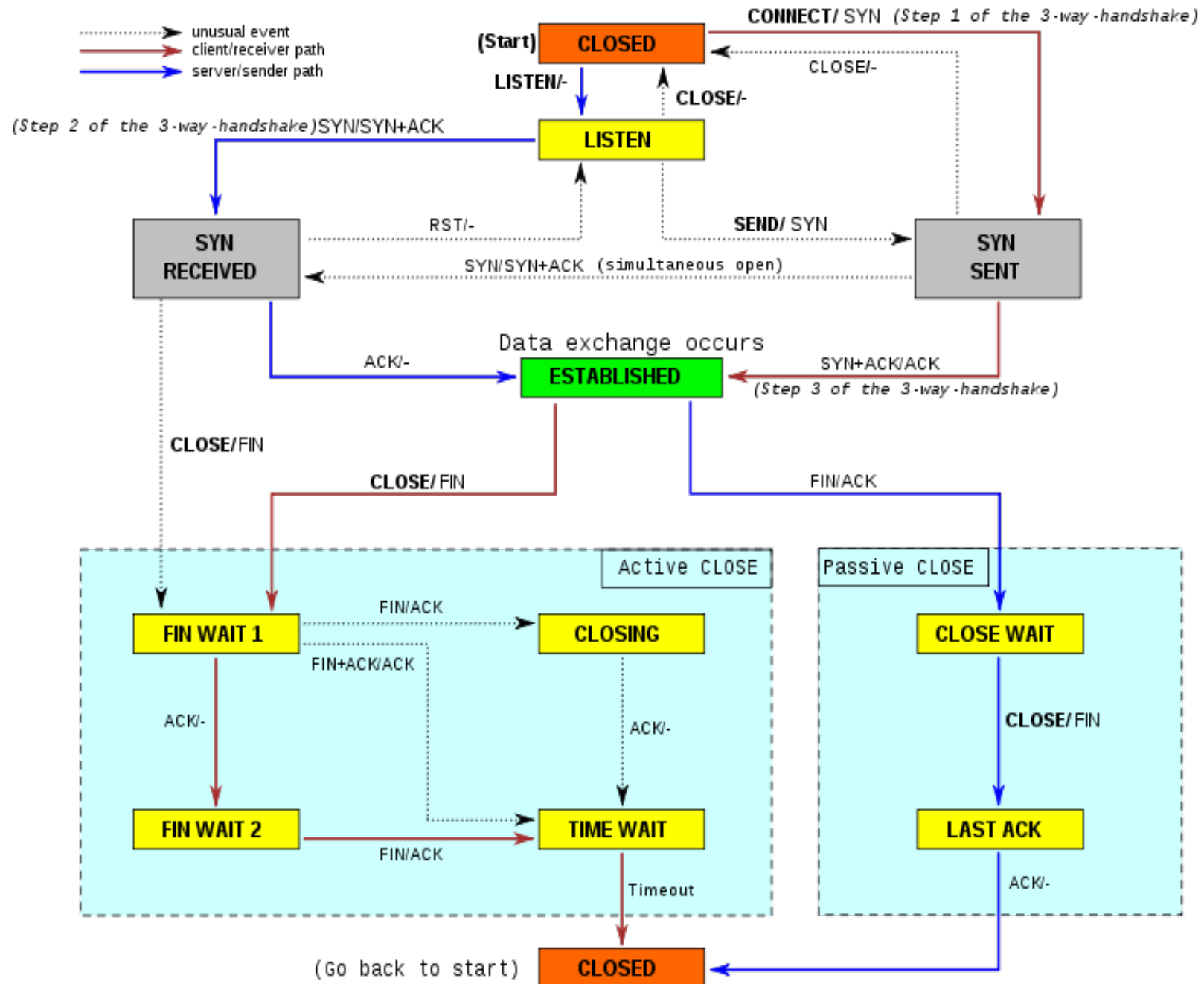
- Connection oriented
- Streams
- Reliability (handles automatically some of the problems of UDP): packets that do not arrive and packets that arrive twice or out of order.
  - Sequence number of the packet (number of bytes sent), if the sequence number is 7200 and the packet size=1024 => new seq\_number = 8224
  - Initialized randomly (to make connection hijacking more difficult)
  - It sends many packets at once (before receiving confirmation). TCP window size (between 2 and 65,535 bytes \* window scaling)
  - The receiver can adjust the window size with flow control (it can slow down or pause the connection) specifying the maximum buffer size that is willing to receive.
  - Network congestion control can also be used
- This is the protocol used normally by: HTTP, SSH, etc.

# 3 way handshake

- Establishing a connection requires 3 packets: SYN (I want to establish comm. and this is the packet's sequence number), SYN-ACK (alright, here is mine, ACK-OK).
- And 3 other packets to end the connection: FIN, FIN-ACK, ACK.
- Or a longer chain of FIN-ACK.



# State diagram



# TCP *versus* UDP

- TCP:
  - Solves some of UDP problems (reliability, network congestion, ...)
  - It's slower than UDP due to the handshake
  - It's not appropriate for real time transmission because it resends lost packets
- UDP:
  - It's faster (it doesn't have as much overhead)
  - It's more difficult to handle in code
  - You can use your own protocol

# TCP is connection oriented

- **connect ( )**
  - In the client side
  - It triggers the handshake protocol and can potentially fail
  - It establishes a permanent connection between 2 sockets (virtual circuit)
- There are 2 types of sockets in the **server side**:
- **listening socket**: used to listen for connections
  - There is only one on the server
  - It has a (listening IP address, listening port)
- **Active socket**: connected to an external client socket with its own (client IP address, client port)
  - There can be many of them on the server (thousands)
  - Each one has a different quadruple:  
(listening IP address, listening port)  $\Leftrightarrow$  (client IP address, client port)
  - The (listening IP address, listening port) is the same for all active sockets
  - They can act as a file for the OS.

# Algorithm to handle a TCP socket on the server:

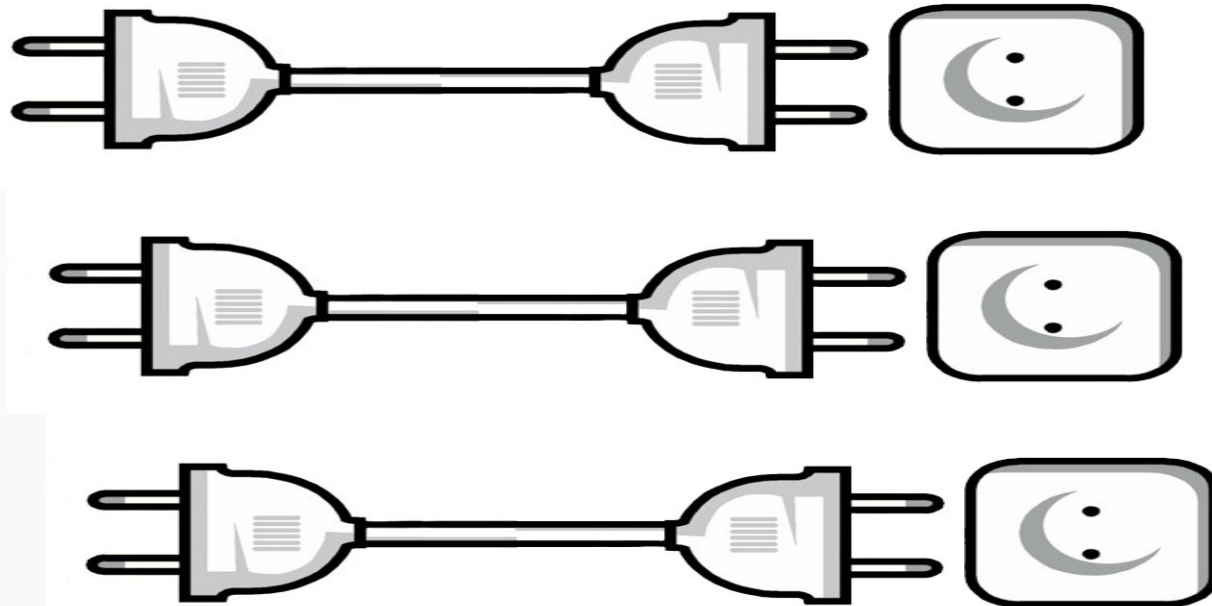
- Create a **listening socket**:
  - `s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
  - `s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`
- We'll listen to hosts on a given port:
  - `s.bind((HOST, PORT))`
  - `s.listen(max. simultaneous connections)`
- Each simultaneous connection can be handled in a different thread or process (we'll see this on a different chapter)
- Infinite loop:
  - Create a connected active socket: `sc, sockname = s.accept()`
  - Data loop:
    - Read data: `data = sc.recv(max_length)`
    - Send reply: `sc.sendall(reply)`
- Close connection (closing handshake): `sc.close()`



# Passive and active sockets



Listening socket  
“s”. Created by  
`socket()` with up to a  
maximum of 3  
simultaneous  
connections.



The different active sockets “sc”  
created by `accept()` : each one has a  
different quadruple

# When is it decided?

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

**s.bind()**

Bind() can either be used on the server or the client (although it's not common) to request a specific port

**Server**

**client**

**s.listen()**

**s.connect()**

It's only used on the server and converts the socket "s" to a listening socket

In the client. It converts the socket "s" to active.

```
sc, sockname = s.accept()
```

In the server. It returns active sockets "sc".

## Algorithm to handle a TCP socket on the client:

- Create a **socket**:
  - **`s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`**
- It assigns the **IP address** from the local machine to “**s**”
- The OS also assigns to “**s**” a **random port number**
- Establish a connection with the server
- Establecer conexión con el servidor:
  - **`s.connect((server_address, server_port))`**
    - Establishes the 3-way handshake between client and server
- Data loop:
  - Send data: **`s.sendall(datos)`**
  - Read reply: **`reply = s.recv(max_length)`**
- Close connection (closing handshake): **`s.close()`**
- Example: tcp\_sixteen.py

# Connection using streams (send)

- The connection is a type of stream with input and output buffers
- This implies that **send()** and **recv()** mean different things in TCP and UDP (where they were atomic functions)
- There are several possibilities when using “**send()**”
- Existen varias posibilidades para “**send ( )**”
  - Either data is accepted by the other end or there is enough space in the output buffer: in both cases, **send()** ends and returns the total length sent
  - Only part of the data is accepted: **send()** ends returning the number of bytes sent
  - If the network card is busy or the output buffer is full, **send()** is blocked while waiting.
- For this reason you have to place **send()** within a loop
- Python provides a method called “**sendall(data)**”

# Connection using streams (recv)

- **recv(number of bytes to read)**
  - If there is no data in the input buffer, it becomes blocked until data arrives
  - If there is enough data in the input buffer, it ends by returning the number of bytes you requested (read bytes)
  - If there is data in the input buffer, but not enough, it also ends but returns how much it contains (read bytes)
- This forces you to place **recv()** inside a loop
- Python has the method **sendall()** but it only provides **recv()** for receiving
- The method **recv\_all()** doesn't exist
- We have a problem when reading:
  - In case we know beforehand the message length, we read as much data
  - If we don't know the length, we can read up to the end of the stream
  - We can also send the message length as part of the message (similarly to the field “**content-length:**” used in HTTP)

# Other methods in the socket class:

- `s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`
  - If it's not set, the OS remains 4 minutes in the state “time wait” (when a socket is closed)
- **`sc.getsockname()`**
  - For both active and listening sockets. It returns its address
- **`sc.getpeername()`**
  - Only for active sockets. It returns its peer's address

# deadlock

- It happens when client/server wait for each other indefinitely
- It can happen very easily with TCP due to the use of an input and output buffer of limited size
  - Output buffer: it stores data until it can be sent
  - Input buffer: reads data until the application can process it
- This doesn't happen with UDP
- Example: **tcp\_deadlock.py**
- This program fills the buffers and then triggers the flow control system (that can either slow down or stop the flow) to prevent the client from sending more data (but the client doesn't read until it has written everything, therefore it gets blocked)
- Possible solutions:
  - **s.setblocking(False)** to disable blocking for **send()** and **recv()** (they will end immediately if they can't read or write)
  - Reading and writing can be divided into different **threads** or **processes**, for example, a thread that writes and a different thread that reads
  - Using **select()**, **poll()** or **epoll()** provided by the OS to obtain (manually) an event-driven program
  - Or the best option which is to use an **event-driven framework** that handles the logic automatically.

# Bidirectional sockets

- Sockets are bidirectional: the same socket can be used to **send()** or to **recv()** data
- Despite the fact that they are not always used in both directions, it's not possible to create a one-way socket
- The solution is to close one direction but not the other:
  - **shutdown(SHUT\_WR)** , **shutdown(SHUT\_RD)** , **shutdown(SHUT\_RDWR)**
- In Python (but not in POSIX sockets):
  - A **socket** can **recv()** and **send()**
  - A **file** can **read()** and **write()**
  - But not both at the same time
  - But we have the method “**makefile()**” that returns a file
  - We also have the method “**fileno()**”



# Review questions

- Which layer does TCP belong to?
- What elements form a TCP/IP address?
- Does TCP have session?
- Why is the sequence number initialized randomly?
- What are the main differences between TCP and UDP?

## Exercise 1: send messages without a specified length

- Write a program in which the client sends a message to the server and it replies with the address and size of the received message.
- You must use TCP sockets
- Do it in two different ways:
  1. Reading with **recv()**, sending data with **sendall()** and signalling the end of the message with **shutdown()**
  2. Obtain a file from the socket to be able to read and write using **read()** and **write()**, you must then send the size of the message first and read the message after that
- Estimated time: 50 minutes

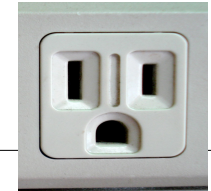
## Exercise 2: proxy

- Using the second technique from the previous exercise. Create a TCP server that receives a message from a client and sends it to a second client that is connected to this server. Once the message is sent to the second client, the connection will end.
- Estimated time: 45 minutes

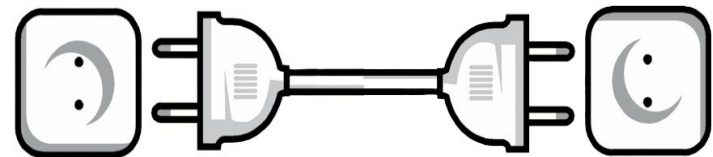
# Recordar que el protocolo IP se encarga de:

- Enrutamiento de los paquetes.
  - Para ello existen las direcciones IP (32 bits =  $4 \times 8$ , en IPv4) que especifican un host (interface con 128 bits =  $8 \times 16$ , en IPV6):
    - 127.\*.\*.\* (IANA reservado)
    - 10.\*.\*.\*, 172.16-31.\*.\*, 192.168.\*.\* (reservados subredes privadas)
    - Localhost = 127.0.0.1 */etc/hosts*
    - Véase /etc/hosts
    - La semana del 3 de febrero de 2011, IANA se quedó sin direcciones IPv4
  - Y los puertos (16 bits) que especifican un proceso dentro del host:
    - 0-1023 (reservado y conocidos. Especial para el S.O.)
    - 1024-49151 (registrados pero no reservados) */etc/services*
    - 49152-65535 (libres)
    - Véase /etc/services
- Fragmentación de los paquetes.

# Qué es un socket



- Un zócalo de enchufe
- Se inventó en el BSD Unix para conectar ordenadores imitando a los archivos
- Hay varias familias: `AF_INET`, `AF_INET6`, `AF_UNIX`
- Y varios tipos en cada familia: `SOCK_STREAM`, `SOCK_DGRAM`
- Para enviar un mensaje se necesitan dos sockets
- Los sockets de un tipo y familia sólo se pueden conectar a los del mismo tipo y familia



# Protocolo UDP (User Datagram Protocol)

- Envía Datagramas
- No utiliza conexión
- Tiene problemas de fiabilidad (*unreliability*):
  - No se garantiza la llegada de los paquetes.
  - Pueden llegar duplicados.
  - Pueden llegar desordenados.
- Para solucionarlo hay que hacer **¡a mano!**:
  - Asignar un número de secuencia para los paquetes (comenzar en random)
  - Un bucle para confirmar la recepción o reenviar el paquete (Backoff)
    - Backoff exponencial.
    - Internet Backoff (con algo de random)
  - Timeouts (si no responde en tiempo)
  - Spoofing (suplantación de identidad)
  - Fragmentación del paquete (MTU)
- Tiene ventajas de ser un protocolo más rápido que el TCP
- Se utiliza mucho en live-streaming media

# Proceso para conectar dos sockets UDP

## Servidor

- crear un socket
- bind((' ', PORT))
- recvfrom()
- sendto()

## Cliente

- crear un socket

- sendto()
- recvfrom()

- connect()
- send()
- recv()

# Ejemplos de clientes/servidor UDP

udp\_local.py

udp\_remote.py

big\_sender.py

udp\_broadcast.py

udp\_server.c

udp\_client.c



# Preguntas de repaso

- ¿Qué es un socket?
- ¿Qué es un protocolo de red?
- ¿Cómo funciona el protocolo UDP?
- ¿Tiene sesión el protocolo UDP?
- ¿Tiene comprobación de errores?
- ¿Se comprueba si se han perdido paquetes o si se han recibido fuera de orden?

## PECS - actividades 13 marzo 2012

- Leer el tema 2 (UDP) del libro de FPNP-2ªed, probando todos los programas.
- Probar los programas en local (servidor y cliente en la misma máquina).
- Probar los programas en remoto (servidor y cliente en máquinas diferentes).
- Investigar el MTU entre ordenadores y en local ¿es el mismo?
- Probar a enviar un mensaje a <BROADCAST> y que lo reciban el resto de los ordenadores.
- Investigar cual es la dirección de broadcast.
- Comunicarse a través de sockets y UDP entre dos ordenadores diferentes pero escribiendo cliente/servidor en dos lenguajes diferentes.

# Entregable, 13 marzo 2012

- Tarea a realizar en grupos de hasta 4 alumnos.
- Escribir un programa cliente y otro programa servidor que se comuniquen mediante sockets por el protocolo UDP. Cliente y Servidor deben estar escritos utilizando dos lenguajes de programación diferentes.
  - a) El cliente enviará al servidor un cierto mensaje de un tamaño sin especificar.
  - b) El servidor muestra, por la salida estándar, la dirección y puerto del cliente, así como el mensaje que este ha enviado.
  - c) El servidor contesta al cliente con un mensaje de respuesta que incluya el tamaño del mensaje original que ha recibido.
  - d) El cliente recibe la respuesta del servidor y muestra, por la salida estándar, la dirección del servidor y el mensaje de respuesta recibido.
  - e) Tanto el cliente como el servidor cierran el socket.
- Tiempo estimado: 30 minutos

# Actividad 2

- Crea un servidor UDP que reciba un mensaje de un cliente en varios paquetes
- En el cliente, puedes tener el mensaje almacenado en una lista de cadenas y trabajar con esta lista. Cada cadena sería una palabra del mensaje.
- Desde el cliente envía primero el número de palabras del mensaje. Cada palabra se enviará en un paquete distinto.
- Antes de enviar cada palabra envía su posición en el mensaje
- Elige aleatoriamente que palabra se envía en cada iteración, no importa que se envíe el mismo paquete (palabra) más de una vez
- El servidor recibe primero la longitud del mensaje en número de palabras, luego antes de recibir cada palabra, recibe su posición en la lista y luego recibe la palabra en sí
- El servidor saldrá del bucle y cerrará el socket cuando tenga el mensaje completo ordenado que deberá mostrar por pantalla

## Actividad 2

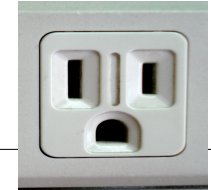
- El ejercicio pretende demostrar un mecanismo para ordenar paquetes y descartar repetidos. Normalmente el protocolo TCP se encarga de esto a bajo nivel y nunca se suele hacer esto desde código Python.
- Utiliza las funciones `sendto()` y `recvfrom()`
- Puedes trabajar con el siguiente mensaje:  

```
mensaje = ["hola", "esto", "es", "un", "mensaje"]
```
- Tiempo estimado: 45 minutos

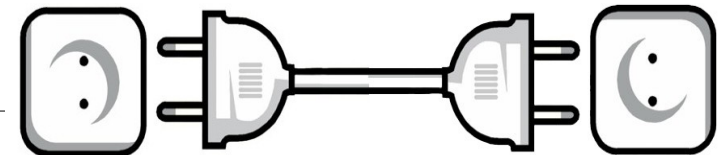
# Remember that the IP protocol is responsible for:

- Packet routing.
  - To do that it uses IP addresses (32 bits =  $4 \times 8$ , in IPv4) that specify a host (128 bits =  $8 \times 16$ , in IPV6):
    - 127.\*.\*.\* (IANA reserved)
    - 10.\*.\*.\*, 172.16-31.\*.\*, 192.168.\*.\* (reserved for private subnets)
    - Localhost = 127.0.0.1 */etc/hosts*
    - See /etc/hosts
    - The 3rd of February of 2011, IANA ran out of IPv4 addresses
  - It also uses ports (16 bits) that specify a process within the host:
    - 0-1023 (reserved well known ports. Specific uses by the OS.)
    - 1024-49151 (registered but not reserved) */etc/services*
    - 49152-65535 (free to use)
    - See /etc/services
- Packet fragmentation

# What is a socket?



- Like a wall socket
- It was invented for BSD Unix to connect computers while imitating file behavior
- There are various families: `AF_INET`, `AF_INET6`, `AF_UNIX`
- And several types within each family: `SOCK_STREAM`, `SOCK_DGRAM`
- To send a message you need two sockets
- Socket of one type and family can only connect to sockets of the same type and family



# UDP (User Datagram Protocol)

- Sends datagrams
- Connectionless
- *Has unreliability issues:*
  - Packet delivery is not guaranteed
  - They can arrive more than once
  - They can arrive out of order
- We must solve these problems **by hand!:**
  - Assign a sequence number to each packet (begin with a random number)
  - A loop to verify reception or to resend a packet (Backoff)
    - Exponential Backoff.
    - Internet Backoff (with some randomness)
  - Timeouts (if no reply is received on time)
  - Spoofing
  - Packet fragmentation (MTU)
- It has the advantage of being faster than TCP
- It's used a lot for streaming live media



# Process to connect two UDP sockets

## Server

- Create a socket
- `bind('', PORT)`
- `recvfrom()`
- `sendto()`

## Client

- Create a socket

- `sendto()`
- `recvfrom()`

- `connect()`
- `send()`
- `recv()`

# UDP client/server examples

udp\_local.py

udp\_remote.py

big\_sender.py

udp\_broadcast.py

udp\_server.c

udp\_client.c

# Review questions

- What's a socket?
- What's a network protocol?
- How does the UDP protocol work?
- Does the UDP protocol have session?
- Does it check for errors?
- Does it check whether packets have been dropped or whether they have been received out of order?

# PECS - exercises 13th March 2012

- Read Chapter 2 (UDP) from the book FPNP-2<sup>a</sup>ed, testing every code listing.
- Test every program locally (server and client on the same machine).
- Test the programs remotely (server and client in different machines).
- Look into the MTU between computers and locally, is it the same?
- Try to broadcast a message so it is received by all the other computers.
- Find out which is the broadcast address.
- Communicate using sockets and UDP between two different computers but writing client and server in two different programs.

# Assignment, 13th March 2012

- Work in groups of up to 4 students.
- Write a client and a server that communicate with UDP using sockets. The client and server must be written two different programming languages.
  - a) The client will send the server a message without specifying its length.
  - b) The server will show in the standard output, the address and port the client is using, as well as the message sent.
  - c) The server replies to the client with a response message that includes the size of the original message it received.
  - d) The client receives the reply from the server and shows in the standard output the address of the server and the response message received.
  - e) Both client and server must close the socket
- Estimated time: 30 minutes

# Exercise 2

- Write a UDP server that receives a message from a client in several packets
- In the client you can have the message stored as a list of strings and work with this list. Each string would be a word in the message
- From the client you first send the number of words the message has. Each word will be sent in a different packet
- Before sending each word, send its position in the message
- Choose randomly which word will be sent in each iteration, it doesn't matter that the same packet (word) is sent more than once
- The server first receives the number of words the message has, then before receiving each word, it receives its position in the list and then receives the word itself
- The server will break out of its loop and will close the socket once the whole message has been received. The message will then be shown on screen.

## Exercise 2

- This exercise is meant to demonstrate a mechanism to order packets and discard repeated ones. Normally the TCP protocol does it at a low level and this is not usually done from Python code
- Use the functions `sendto()` and `recvfrom()`
- You can use the following message:  

```
message = ["hi", "this", "is", "a", "message"]
```
- Estimated time: 45 minutes

# Programación en Entornos Cliente/Servidor

Resultado de la encuesta realizada el 29 de mayo de 2012

|                                         | 1   | 2   | 3   | 4   | 5    | 6   | 7   | 8   | 9   | 10  | 11  | 12  | 13  | 14  | 15  | 16  | 17  | 18   |
|-----------------------------------------|-----|-----|-----|-----|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
| La asignatura me ha gustado             | 7,5 | 7,5 | 2,5 | 7,5 | 10   | 7,5 | 7,5 | 7,5 | 5   | 5   | 7,5 | 7,5 | 7,5 | 7,5 | 7,5 | 5   | 5   | 7,5  |
| Creo que puede servirme en mi profesión | 7,5 | 5   | 10  | 10  | 10   | 5   | 7,5 | 7,5 | 7,5 | 7,5 | 5   | 7,5 | 5   | 5   | 7,5 | 7,5 | 5   | 7,5  |
| El temario es adecuado                  | 7,5 | 5   | 7,5 | 7,5 | 7,5  | 5   | 7,5 | 5   | 7,5 | 7,5 | 5   | 5   | 5   | 5   | 7,5 | 7,5 | 5   | 7,5  |
| El profesor domina el tema              | 7,5 | 7,5 | 5   | 5   | 10   | 7,5 | 7,5 | 7,5 | 7,5 | 10  | 7,5 | 7,5 | 7,5 | 5   | 7,5 | 7,5 | 7,5 | 10   |
| El profesor se expresa con claridad     | 7,5 | 7,5 | 7,5 | 5   | 10   | 7,5 | 10  | 5   | 5   | 7,5 | 7,5 | 7,5 | 7,5 | 5   | 7,5 | 7,5 | 7,5 | 10   |
| La metodología docente es adecuada      | 7,5 | 7,5 | 2,5 | 7,5 | 5    | 5   | 5   | 5   | 2,5 | 5   | 5   | 5   | 5   | 2,5 | 5   | 5   | 5   | 7,5  |
| Carga de trabajo adecuada               | 5   |     | 7,5 | 2,5 | 7,5  | 5   | 7,5 | 5   | 5   | 5   | 5   | 5   | 5   | 2,5 | 2,5 | 2,5 |     | 7,5  |
| Carga de trabajo bien repartida         | 7,5 | 5   | 7,5 | 5   | 7,5  | 5   | 5   | 5   | 5   | 5   | 5   | 5   | 5   | 2,5 | 5   | 2,5 | 7,5 | 7,5  |
| Organización en actividades me gusta    | 7,5 | 5   | 2,5 | 5   | 10   | 5   | 5   | 2,5 | 5   | 2,5 | 7,5 | 5   | 0   | 0   | 7,5 | 5   | 7,5 | 5    |
| Preferiría un enfoque más teórico       | 0   | 0   | 0   | 0   | 0    | 2,5 | 7,5 | 2,5 | 2,5 | 2,5 | 2,5 | 0   | 0   | 0   | 0   | 0   | 0   | 0    |
| Creo que se debería dar más materia     | 10  | 0   | 0   | 0   | 7,5  | 0   | 5   | 5   | 2,5 | 2,5 | 0   | 0   | 0   | 0   | 7,5 | 0   | 0   | 0    |
| Me gustan las asignaturas teóricas      | 0   | 0   | 0   | 0   | 0    | 2,5 | 7,5 | 2,5 | 5   | 5   | 0   | 0   | 2,5 | 2,5 | 0   | 0   | 0   | 0    |
| Me gustan las asignaturas prácticas     | 10  | 10  | 10  | 10  | 10   | 7,5 | 2,5 | 10  | 10  | 10  | 10  | 7,5 | 10  | 10  | 10  | 10  | 10  | 10   |
| nº horas                                | 3   | 4,5 | 3,5 | 3,5 | 4    | 2   | 3   | 2   | 1   | 2   | 2   | 2   | 2   | 2   | 2   | 2   | 2   | 2    |
| de acuerdo con la asignatura            | 7,5 | 5,8 | 6,7 | 8,3 | 9,2  | 5,8 | 7,5 | 6,7 | 6,7 | 6,7 | 5,8 | 6,7 | 5,8 | 5,8 | 7,5 | 6,7 | 5,0 | 7,5  |
| de acuerdo con el profesor              | 7,5 | 7,5 | 6,3 | 5,0 | 10,0 | 7,5 | 8,8 | 6,3 | 6,3 | 8,8 | 7,5 | 7,5 | 7,5 | 5,0 | 7,5 | 7,5 | 7,5 | 10,0 |
| de acuerdo con la metodología           | 6,9 | 5,8 | 5,0 | 5,0 | 7,5  | 5,0 | 5,6 | 4,4 | 4,4 | 4,4 | 5,6 | 5,0 | 3,8 | 1,9 | 5,0 | 3,8 | 6,7 | 6,9  |

media

|                                         |     |     |
|-----------------------------------------|-----|-----|
| La asignatura me ha gustado             | 6,6 | 6,8 |
| Creo que puede servirme en mi profesión | 7,1 |     |
| El temario es adecuado                  | 6,6 |     |
| El profesor domina el tema              | 7,2 | 6,9 |
| El profesor se expresa con claridad     | 6,6 |     |
| La metodología docente es adecuada      | 5,4 | 5,3 |
| Carga de trabajo adecuada               | 5,3 |     |
| Carga de trabajo bien repartida         | 5,5 |     |
| Organización en actividades me gusta    | 5,1 |     |
| Preferiría un enfoque más teórico       | 1,3 |     |
| Creo que se debería dar más materia     | 3,0 |     |
| Me gustan las asignaturas teóricas      | 2,4 |     |
| Me gustan las asignaturas prácticas     | 8,8 |     |
| nº horas                                | 2,3 |     |



| 19  | 20   | 21  | 22  | 23  | 24  | 25  | 26  | 27  | 28  | 29  | 30  | 31   | 32  | 33  | 34  | 35  | 36  | 37  | 38  | media |
|-----|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|-----|-----|-----|-----|-----|-----|-----|-------|
| 5   | 10   | 5   | 5   | 7,5 | 5   | 5   | 5   | 7,5 | 5   | 10  | 7,5 | 10   | 7,5 | 7,5 | 0   | 7,5 | 7,5 | 7,5 | 5   | 6,6   |
| 7,5 | 10   | 5   | 5   | 10  | 7,5 | 7,5 | 5   | 7,5 | 5   | 10  | 10  | 10   | 7,5 | 7,5 | 5   | 7,5 | 7,5 | 5   | 2,5 | 7,1   |
| 7,5 | 10   | 7,5 | 5   | 10  | 7,5 | 7,5 | 5   | 5   | 7,5 | 7,5 | 10  | 10   | 5   | 5   | 2,5 | 5   | 7,5 | 5   | 5   | 6,6   |
| 7,5 | 10   | 7,5 | 5   | 7,5 | 5   | 5   | 7,5 | 7,5 | 5   | 10  | 10  | 5    | 7,5 | 7,5 | 0   | 7,5 | 7,5 | 7,5 | 7,5 | 7,2   |
| 7,5 | 10   | 7,5 | 2,5 | 5   | 5   | 5   | 7,5 | 5   | 5   | 7,5 | 7,5 | 5    | 7,5 | 7,5 | 0   | 5   | 7,5 | 5   | 7,5 | 6,6   |
| 5   | 10   | 5   | 5   | 7,5 | 5   | 5   | 5   | 2,5 | 5   | 10  | 10  | 2,5  | 7,5 | 5   | 2,5 | 5   | 5   | 2,5 | 7,5 | 5,4   |
| 5   | 10   | 7,5 | 5   | 5   | 5   | 5   | 7,5 | 5   | 0   | 7,5 | 7,5 | 5    | 5   | 5   | 5   | 7,5 | 5   | 0   | 7,5 | 5,3   |
| 2,5 | 10   | 5   | 5   | 5   | 7,5 | 7,5 | 7,5 | 5   | 2,5 | 10  | 7,5 | 0    | 7,5 | 5   | 2,5 | 7,5 | 5   | 2,5 | 7,5 | 5,5   |
| 5   | 10   | 7,5 | 2,5 | 7,5 | 5   | 5   | 5   | 5   | 2,5 | 10  | 7,5 | 0    | 10  | 7,5 | 5   | 2,5 | 5   | 0   | 5   | 5,1   |
| 0   | 2,5  | 5   | 2,5 | 0   | 2,5 | 2,5 | 0   | 0   | 0   | 5   | 7,5 | 0    | 0   | 0   | 2,5 | 0   | 0   | 0   | 0   | 1,3   |
| 2,5 | 5    | 5   | 2,5 | 2,5 | 2,5 | 2,5 | 0   | 0   | 0   | 10  | 7,5 | 10   | 0   | 2,5 | 10  | 2,5 | 5   | 0   |     | 3,0   |
| 0   | 2,5  | 7,5 | 2,5 | 5   | 5   | 5   | 0   | 2,5 | 0   | 7,5 | 10  | 2,5  | 2,5 | 0   | 5   | 0   | 0   | 2,5 | 5   | 2,4   |
| 10  | 10   | 10  | 5   | 10  | 5   | 5   | 7,5 | 7,5 | 10  | 10  | 10  | 10   | 7,5 | 10  | 5   | 10  | 10  | 5   | 7,5 | 8,8   |
| 2   |      | 2   | 2   | 3   | 2   | 1   | 2   | 3   | 2   | 2   | 2   | 6    | 2,5 | 2   | 0,5 | 2   | 2   | 2   | 1,5 | 2,3   |
| 6,7 | 10,0 | 5,8 | 5,0 | 9,2 | 6,7 | 6,7 | 5,0 | 6,7 | 5,8 | 9,2 | 9,2 | 10,0 | 6,7 | 6,7 | 2,5 | 6,7 | 7,5 | 5,8 | 4,2 | 6,8   |
| 7,5 | 10,0 | 7,5 | 3,8 | 6,3 | 5,0 | 5,0 | 7,5 | 6,3 | 5,0 | 8,8 | 8,8 | 5,0  | 7,5 | 7,5 | 0,0 | 6,3 | 7,5 | 6,3 | 7,5 | 6,9   |
| 4,4 | 10,0 | 6,3 | 4,4 | 6,3 | 5,6 | 5,6 | 6,3 | 4,4 | 2,5 | 9,4 | 8,1 | 1,9  | 7,5 | 5,6 | 3,8 | 5,6 | 5,0 | 1,3 | 6,9 | 5,3   |

# Cachés, Message Queues, y Map-Reduce

- De nuevo tenemos un capítulo donde vamos a tratar, de forma rápida, varias tecnologías.
  - MemCached
  - Sharding y funciones Hash
  - Message Queues
  - Map-Reduce
- Cualquiera de ellas tiene libros dedicados por entero.
- La idea es conocer su existencia y para qué se utilizan.

# Arquitectura de Memcached de wikipedia

- El sistema usa una arquitectura **cliente-servidor**
  - Los servidores almacenan un diccionario
    - Las claves son de hasta 250 bytes y los valores pueden ser como mucho de 1 megabyte
  - Los clientes rellenan el diccionario y realizan consultas
    - Los clientes utilizan **bibliotecas** para contactar con el servidor, que por defecto, ofrecerá su servicio en el **puerto 11211**
- Cada cliente conoce todos los servidores, pero los servidores no se comunican entre ellos
- Si un cliente desea establecer o leer el valor correspondiente a cierta clave, la biblioteca del cliente primero calcula el hash de la clave para determinar el servidor que se usará. Luego contacta con el servidor.
- El servidor calculará un segundo hash de la clave para determinar donde guardar o leer el valor correspondiente
- Los servidores almacenan los valores en RAM.
  - Si un servidor se queda sin memoria, descarta los valores más antiguos
  - Por lo tanto, los clientes deben utilizar Memcached como una cache no permanente
- MemcacheDB es un producto compatible con Memcached que proporciona almacenamiento persistente
- Un despliegue típico tendrá varios servidores y muchos clientes.
  - Sin embargo, es posible utilizar Memcached en un solo ordenador, de forma que actúa simultáneamente como cliente y servidor

# Preguntas de repaso

- ¿Qué es Memcached? ¿Por qué disminuye el tiempo de respuesta de las páginas web?
- ¿Existe Memcached solo para Python?
- ¿Qué es una función hash? ¿Para qué se usan?
- ¿Qué es una cola de mensaje?
- ¿Qué es AMQP? ¿Qué tipos de enrutamiento permite?
- ¿Qué es ØMQ? ¿En qué se diferencia de TCP?
- ¿Cuáles son las distintas topologías que permite ØMQ?
- Explica map-reduce con tus propias palabras

# Memcached de wikipedia

- Es un sistema de memoria distribuida de propósito general
  - Se usa frecuentemente para acelerar las páginas web que utilizan bases de datos dinámicas. Lo hace creando una cache de datos y objetos en memoria RAM para reducir el número de veces que hay que acceder a un recurso externo (ej. una BD)
  - Memcached funciona en Unix, Windows y MacOS. Además está distribuido con licencia BSD.
- La API de Memcached proporciona una tabla hash muy grande distribuida entre muchas máquinas.
  - Cuando la tabla está llena, las inserciones posteriores provocan que los datos más antiguos se eliminen de la tabla en orden LRU (el menos usado recientemente)
- Las aplicaciones que usan Memcached normalmente tienen capas de peticiones que se utilizan antes de recurrir a un medio más lento, como una base de datos
- Es el sistema usado por páginas web como YouTube, Reddit, Zynga, Facebook y Twitter.
  - Google App Engine ofrece un servicio de memcached a través de una API
  - Memcached además funciona con CMSs conocidos como Drupal, Joomla y WordPress

# Ejemplo de utilización de Memcached from wikipedia

- **Seudocódigo** de acceso a una base de datos: primero tal y como sería sin memcached

```
function get_foo(int userid) {
 result = db_select("SELECT * FROM users WHERE userid = ?", userid);
 return result;
}
```

- **Seudocódigo** de acceso a una base de datos: y ahora utilizando memcached

```
function get_foo(int userid) {
 /* first try the cache */
 data = memcached_fetch("userrow:" + userid);
 if (!data) {
 /* not found : request database */
 data = db_select("SELECT * FROM users WHERE userid = ?", userid);
 /* then store in cache until next get */
 memcached_add("userrow:" + userid, data);
 }
 return data;
}
```

# Para utilizar Memcached

- Primero tienes que instalar y ejecutar el servidor (demonio “**memcached**”) en cada uno de tus sistemas:
  - Por ejemplo a través del **gestor de paquetes synaptics**
  - Puedes configurarlo o utilizarlo como viene por defecto
- Luego tienes que instalar un cliente del servidor (demonio) para tu lenguaje de programación favorito:
  - Existen para muchos lenguajes: C/C++, PHP, Java, Python, Ruby, Perl, Windows/.NET, MySQL, PostgreSQL, Erlang, Lua, Lisp, OCaml, etc.
  - **\$ pip install python-memcached**



## Ejemplo de programa Python que utiliza Memcached

- El programa `squares.py` calcula el cuadrado de muchos enteros del 0 al 5000 y los memoriza para no tener que volver a calcularlos.
- Su ejecución proporciona la siguiente salida:
  - `$ ./squares.py`
  - Diez ejecuciones sucesivas:
  - 2.18s 1.56s 1.15s 0.85s 0.62s 0.53s  
0.44s 0.38s 0.35s 0.31s



# Otras características

- Puedes poner una fecha de caducidad a las entradas de memcached.
  - Después de dicha fecha se eliminan del caché automáticamente.
- Puedes eliminar entradas concretas o todo el caché.
- Puedes reescribir (reemplazar) entradas inválidas con otras actualizadas.
- En Python es habitual “decorar” funciones para adaptarlas a memcached.

# Actividad 1: memcached en Python

- Instale el servidor (demonio) y el cliente para Python de **memcached**. Luego ejecute el programa **squares.py** y compruebe que funciona.
- Reescriba el programa **squares.py** con **decoradores**. Véase <http://www.artima.com/weblogs/viewpost.jsp?thread=240808> y las transparencias de Python **clases.pdf**. Nota: esta parte de la actividad hay que realizarla al final de todas las demás actividades.

# Memcached y Sharding

- Ya dijimos que memcached puede utilizar muchos servidores que no se comunican entre sí.
- El cliente debe utilizar una función **hash** para calcular qué servidor consultar.
- Por su lado, memcached también utiliza una tabla **hash** para guardar los datos.
- Las funciones *hash* son muy utilizadas con muchos propósitos:
  - Firma digital
  - Evitar la adulteración (*tampering*) de un archivo
  - Distribuir una base de datos (*sharding*)
  - Etc.
- **Sharding** es un término utilizado en Bases de Datos para indicar la partición de una base de datos entre muchos servidores diferentes.
  - La partición se realiza por **filas** (y no por columnas, como en la normalización)
- A cada partición individual se denomina un **shard** o **database shard**.

# Ejemplo de función *hash* en Python:

- El programa `listaHash.py` utiliza una tabla *hash* para almacenar el diccionario de español que está en `/usr/share/dict/`
- Se puede probar con variar funciones *hash* diferentes
- La calidad de cada función *hash* se puede calcular como el ratio con el caso ideal:
  - Si tenemos  $N$  elementos distribuidos en  $M$  listas.
  - Cada lista de longitud  $L_i$  para  $i \in \text{range}(0, M)$
  - $\sum_i (1 + L_i) * L_i / 2$  (caso real)
  - $(M + N) * N / (2 * M)$  (caso ideal)

## Actividad 2: función hash

- Invéntese una función hash que mejore a la de xor y pruébela con el diccionario de español
- trabajo individual

# Colas de Mensajes

- Las colas de mensajes y buzones son componentes utilizados en la ingeniería del software para la comunicación entre procesos o entre hilos dentro del mismo proceso.
- Utilizan una cola para enviar mensajes – para pasar el control de uno a otro o enviar contenido
- Las colas de mensajes proporcionan un protocolo de comunicación asíncrono:
  - Esto significa que el emisor y receptor del mensaje no necesitan interactuar con la cola de mensajes al mismo tiempo.
  - Los mensajes de la cola se almacenan hasta que el receptor los lea.
- Se trata de un tipo especial de “Message Oriented Middleware”

# Message-oriented middleware (MOM)

- Consiste en infraestructura de software o hardware que se encarga de enviar y recibir mensajes entre sistemas distribuidos.
- MOM permite que los módulos de una aplicación estén distribuidos entre varias plataformas.
- Reduce la complejidad de desarrollar aplicaciones que abarcan muchos sistemas operativos y protocolos de red
  - Esto lo hace manteniendo al desarrollador de la aplicación al margen de los detalles de los sistemas operativos e interfaces de red
- MOM normalmente proporciona APIs para distintos sistemas y redes
- MOM es software que reside en ambas partes de una arquitectura cliente-servidor
- Y normalmente tiene soporte para llamadas asíncronas entre el cliente y el servidor
- MOM ayuda a que los desarrolladores no pierdan el tiempo con la complejidad del mecanismo maestro-esclavo que utiliza la arquitectura cliente-servidor

# Advanced Message Queuing Protocol (AMQP)

- Es un protocolo de estándar abierto de la capa de aplicación para middleware orientada a mensajes.
- Las características que definen AMQP son orientación a mensajes, colas, enrutamiento (incluyendo punto a punto y publicación-suscripción), confiabilidad y seguridad
- AMQP permite la interoperabilidad entre las implementaciones de distintos proveedores
  - Del mismo modo que SMTP, HTTP, FTP, etc... permiten la interoperabilidad de sistemas
- Existen varias implementaciones de MOM como:
  - Apache Active MQ.
  - Rabbit MQ (escrito en Erlang, implementa AMQP)
  - Zero MQ (ØMQ) es una versión “light” y rápida de la anterior en forma de librería.



# ØMQ en cien palabras

- ØMQ (ZeroMQ, 0MQ, zmq) parece una biblioteca de red que se puede incorporar en programas
- Pero en realidad actúa como un **framework de concurrencia**
- Proporciona **sockets** que transportan mensajes a través de distintos medios: entre procesos, internamente dentro de un proceso, TCP y multicast
- Puedes conectar sockets **N-a-N** con patrones como **fanout**, **pub-sub**, **distribución de tareas** y **petición-respuesta**.
- Es suficientemente **rápido** como para ser la fibra de clústeres
- Su modelo de E/S asíncrono permite crear aplicaciones multinúcleo escalables creadas como **tareas de procesamiento de mensajes asíncronas**
- Its asynchronous I/O model gives you scalable multicore applications, built as **asynchronous message-processing tasks**.
- Tiene **APIs** para muchos lenguajes y es compatible con la mayoría de los sistemas operativos
- ØMQ pertenece a iMatix y tiene licencia de software libre **LGPL**

# ØMQ?

- Son sockets pero mucho mejor
- Como buzones con enrutamiento
- ¡Es muy rápido!
  - Vivimos en un mundo conectado y el software de hoy en día tiene que navegar por este mundo
  - Por esta razón, los componentes básicos de los grandes programas del futuro van a estar conectados y funcionarán en paralelo
  - Ya no vale solo con que el código sea bueno y que no haga ruido
  - El código de un programa tiene que hablar con el de otro
  - El código tiene que ser sociable, extrovertido y tener muchos amigos.
  - El código tiene que funcionar como la mente humana, billones de neuronas individuales enviando mensajes una a la otra, una red paralela gigante sin un control centralizado, ningún punto débil y aun así capaz de resolver problemas muy difíciles
  - No es por accidente que en el futuro el código se asemejará a la mente humana, porque los extremos de todas las redes son en cierto sentido mentes humanas.

# Lo que intentan hacer y cómo se utiliza

- Para arreglar el mundo, necesitamos hacer dos cosas:
  - Primero, resolver el problema general de “como conectar cualquier fragmento de código a cualquier otro fragmento en cualquier parte”.
  - Segundo, encapsular todo eso en los componentes más básicos que la gente pueda entender y usar fácilmente
- ¡ Quieren cambiar la forma de programar !
- Para utilizarlo hay que:
  - **Instalar el paquete que contiene la librería dinámica**  
(descargarlo desde su página web)
  - E instalar **el cliente para Python** (o el lenguaje que se desee) a través de `$ pip install pzmq-static`

# Ejemplo de utilización de ØMQ: servidor en C++

- Sigue un servidor y un cliente en C++ (programa “holamundo”)

```
//
// Hello World server in C++
// Binds REP socket to tcp://*:5555
// expects "Hello" from the client, replies with "World"

#include <zmq.hpp>
#include <string>
#include <iostream>
#include <unistd.h>

int main() {
 // prepare our context and socket
 zmq::context_t context(1);
 zmq::socket_t socket(context, ZMQ_REP);
 socket.bind("tcp://*:5555");

 while(true) {
 zmq::message_t request;

 // wait for next request from client
 socket.recv(&request);
 std::cout << "Received Hello" << std::endl;

 // do some 'work'
 sleep(1);

 // Send reply back to client
 zmq::message_t reply(5);
 memcpy ((void *) reply.data (), "World", 5);
 socket.send(reply);
 }
 return 0;
}
```

# Ejemplo de utilización de ØMQ: cliente en C++

```
//
// Hello World client in C++
// Connects REQ socket to tcp://localhost:5555
// sends "Hello" to server, expects "World" back
//

#include <zmq.hpp>
#include <string>
#include <iostream>

int main()
{
 // prepare our context and socket
 zmq::context_t context(1);
 zmq::socket_t socket(context, ZMQ_REQ);

 std::cout << "Connecting to Hello World server..." << std::endl;
 socket.connect("tcp://localhost:5555");

 // Do 10 requests, waiting each time for a response
 for (int request_nbr=0; request_nbr != 10; request_nbr++) {
 zmq::message_t request(6);
 memcpy((void *) request.data(), "Hello", 5);
 std::cout << "Sending Hello " << request_nbr << "..." << std::endl;
 socket.send(request);

 // get the reply
 zmq::message_t reply;
 socket.recv(&reply);
 std::cout << "Received World " << request_nbr << std::endl;
 }
 return 0;
}
```

# Compilando y ejecutando ejemplo en C++ ...

- `$ g++ -o zmq_server helloWorld_zmq_server.c -lzmq`
- `$ g++ -o zmq_client helloWorld_zmq_client.c -lzmq`

# Ejecutando el servidor:

```
•$ export LD_LIBRARY_PATH=/usr/local/lib
•$./zmq_server
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
•Received Hello
```

# Ejecutando el cliente:

```
•$ export LD_LIBRARY_PATH=/usr/local/lib
•$./zmq_client
•Connecting to Hello World server...
•Sending Hello 0...
•Received World 0
•Sending Hello 1...
•Received World 1
•Sending Hello 2...
•Received World 2
•Sending Hello 3...
•Received World 3
•.....
```

## Actividad 3: ejemplo de ØMQ en C++

- Instalar la librería de ZMQ: desde la página web `http://www.zeromq.org/`
- Escribir y ejecutar los programas cliente y servidor en C++

# El API de Sockets

- Para crear y cerrar sockets utiliza las funciones **zmq\_socket(3)** y **zmq\_close(3)**.
- Para cambiar y consultar las opciones de configuración de sockets utiliza **zmq\_setsockopt(3)** y **zmq\_getsockopt(3)**
- Para conectar los sockets a la topología de red creando conexiones ØMQ utiliza las funciones **zmq\_bind(3)** y **zmq\_connect(3)**.
- Para enviar y recibir datos a través de los sockets utiliza **zmq\_send(3)** y **zmq\_recv(3)**.



# Las conexiones de ØMQ son distintas de las conexiones típicas de TCP

- Utilizan cualquier modo de transporte (internamente dentro de un proceso, entre procesos, tcp, pgm o epgm)
- La conexión se crea cuando un cliente hace `zmq_connect(3)` a un extremo, independientemente de si el servidor ya ha llamado `zmq_bind(3)` para ese extremo
- Son asíncronas y utilizan colas que aparecen por arte de magia donde y cuando hagan falta.
- Pueden mostrar cierto patrón de envío de mensajes, dependiendo del tipo de socket utilizado en cada extremo.
- Un socket puede tener muchas conexiones entrantes y muchas salientes.
- No existe el método `zmq_accept()`
  - Cuando un socket se enlaza con un extremo, automáticamente empieza a aceptar conexiones
- El código de tu aplicación no puede trabajar directamente con estas conexiones; están encapsuladas por el socket

# Los sockets tienen tipo

- Un servidor que actúa como nodo puede enlazarse a varios extremos y puede hacer esto utilizando un único socket. Esto significa que aceptará conexiones por más de un sitio.
- Un cliente que actúa como nodo también puede conectarse con varios extremos utilizando un solo socket.
- Los sockets tienen tipo.
- El tipo de socket determina la semántica del socket, sus políticas de enrutamiento interno y externo de mensajes, colas, etc.
- Puedes conectar ciertos tipos de socket entre sí, por ejemplo, un socket publicador y otro suscriptor
- Los sockets funcionan conjuntamente utilizando “patrones de mensajes”
- Con ØMQ, defines la arquitectura de la red conectando piezas como si fuese un puzzle.

# Tipos de ØMQ sockets y topologías (1)

- **Patrón petición-respuesta:** se utiliza para enviar peticiones desde un cliente to una o más instancias de un servicio y luego recibir las respuestas por cada petición hecha.
  - **ZMQ\_REQ:** lo utiliza el cliente para enviar peticiones y recibir repsuestas de un servicio
  - **ZMQ\_REP:** lo utiliza el servicio para recibir peticiones y enviar respuestas a un cliente.
  - **ZMQ DEALER:** es un patrón avanzado utilizado como extensión para los sockets de petición/respuesta.
  - **ZMQ\_ROUTER:** es otro patrón avanzado utilizado como extensión para sockets de petición/respuesta

# Tipos de ØMQ sockets y topologías (2)

- **Patrón publicación-suscripción:** se utiliza para la distribución de datos de uno a muchos desde un solo publicador a múltiples suscriptores.
  - **ZMQ\_PUB:** lo utiliza el publicador para distribuir datos.
    - Los mensajes enviados se distribuyen a todos los suscriptores conectados.
  - **ZMQ\_SUB:** lo utiliza el suscriptor para suscribirse a los datos distribuidos por el publicador.
    - Inicialmente un socket ZMQ\_SUB no está suscrito a ningún mensaje. Utiliza la opción ZMQ\_SUBSCRIBE de `zmq_setsockopt(3)` para especificar a que mensajes quieres que suscriba.

# Tipos de ØMQ sockets y topologías (3)

- **Patrón tubería:** se utiliza para distribuir datos a nodos organizados en forma de tubería
  - **ZMQ\_PUSH:** lo utiliza un nodo de la tubería para enviar mensajes a otros nodos colocados después que él en la tubería.
    - Se hace un balanceo de carga entre estos nodos
  - **ZMQ\_PULL:** lo utiliza un nodo de la tubería para recibir mensajes de otros nodos colocados antes que él en la tubería.
    - Se crea una cola con los mensajes de estos nodos.

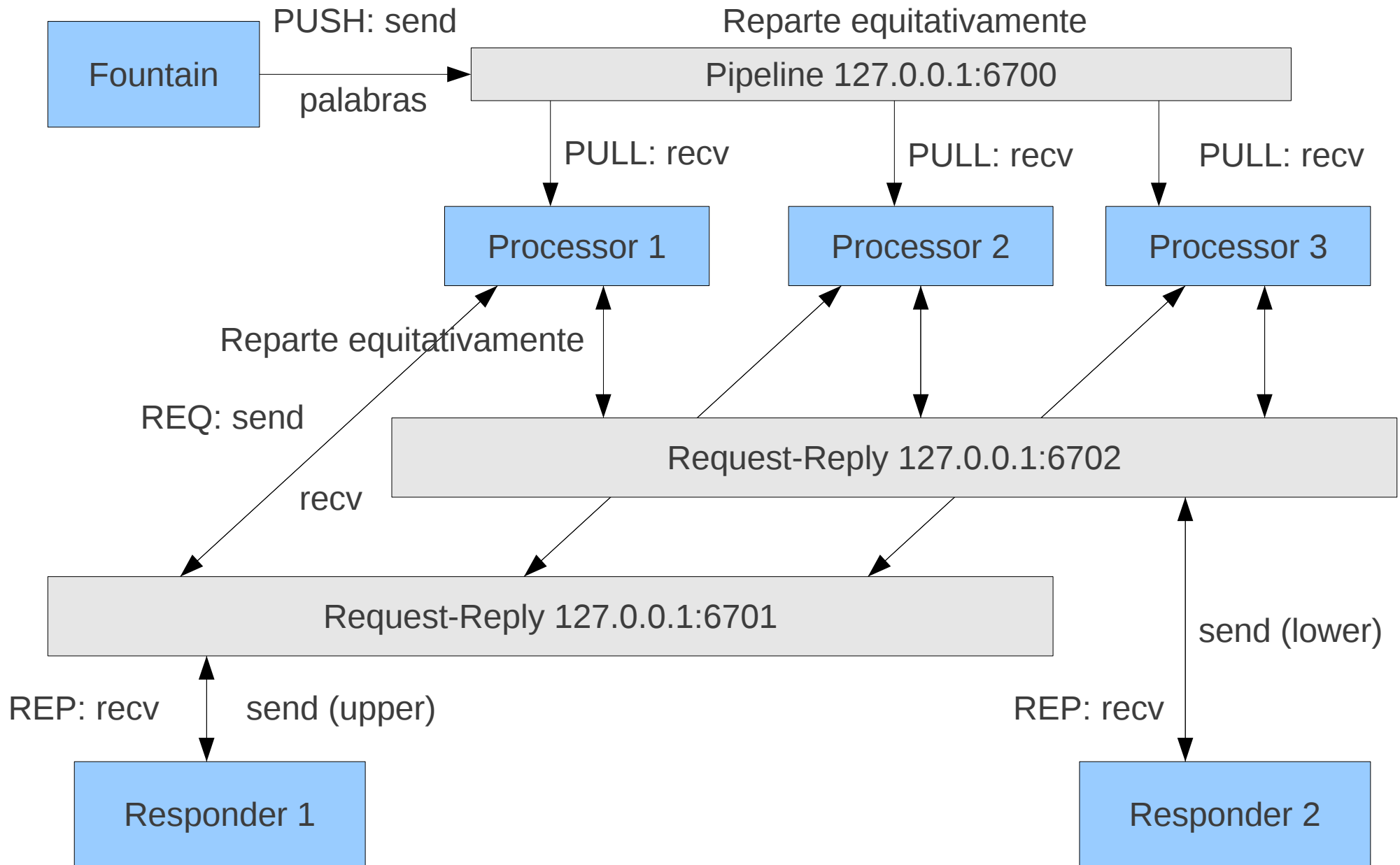
# Tipos de ØMQ sockets y topologías (4)

- **Patrón de pares exclusivos:** se utiliza para conectar pares, las conexiones son solo de 1 a 1.
  - Este patrón se utiliza para comunicación entre hilos dentro de un mismo proceso.
- **ZMQ\_PAIR:** un socket de este tipo solo se puede conectar a un extremo a la vez.
  - No se realiza ningún enrutamiento o filtrado sobre los mensajes que se envían a través de un socket ZMQ\_PAIR

## Actividad 4: Ejemplo de ØMQ en Python

- Instalar el cliente de Python:
  - `$ pip install pzmq-static`
- Escribir y ejecutar el programa `queuecrazy.py` de la página 132 del libro.

# Topología creada por el programa:





# Map-reduce

- Es un framework patentado por Google creado para ayudar con la computación distribuida de conjuntos de datos muy grandes utilizando clústeres.
- MapReduce es un framework para procesar conjuntos de datos grandes relacionados con algunos tipos de problemas distribuibles utilizando un gran número de ordenadores (nodos),
  - Colectivamente se conocen como un clúster (si todos los nodos utilizan el mismo hardware)
  - O como grid (red) si los nodos usan hardware diferente.
- Existen bibliotecas de MapReduce para C++, C#, Erlang, Java, Ocaml, Perl, Python, Ruby, F#, R y otros lenguajes de programación.
- MapReduce es útil para una gran variedad de aplicaciones y arquitecturas, incluyendo: “grep distribuido, ordenación distribuida, inversión de un grafo de enlaces web, estadísticas del log de acceso a una web, construcción de un índice invertido, agrupación de documentos, aprendizaje automático, traducción automática utilizando estadística...”
- En Google, se usó MapReduce para regenerar su índice de la World Wide Web completamente.
  - Reemplazó los programas ad-hoc antiguos que actualizaban el índice y realizaban distintos análisis

# El paso Map y el paso Reduce

- El framework se basa en las funciones map y reduce que se utilizan a menudo en la **programación funcional**
  - Aunque su propósito en el framework no es el mismo que en su forma original
- **El paso "Map"**: el nodo maestro toma la entrada, la divide en sub-problemas más pequeños y los distribuye a nodos que realizan el trabajo.
  - Estos nodos pueden hacer lo mismo, creando una estructura de árbol multi nivel
  - Los nodos de niveles inferiores procesan el problema de menor tamaño y pasan la respuesta de vuelta a su nodo maestro.
- **El paso "Reduce"**: Luego el nodo maestro toma las respuestas de todos los sub-problemas y las combina de forma que obtiene una salida --- la respuesta al problema que trataba de resolver al principio

# Apache Hadoop

- Apache Hadoop es un framework de software que soporta aplicaciones distribuidas bajo una licencia libre.
- Permite a las aplicaciones trabajar con miles de nodos y petabytes de datos.
- Hadoop se inspiró en los documentos Google MapReduce y Google File System (GFS).
- Hadoop es un proyecto de alto nivel Apache que está siendo construido y usado por una comunidad global de contribuidores, mediante el lenguaje de programación Java.
- Yahoo! ha sido el mayor contribuidor al proyecto, y usa Hadoop extensivamente en su negocio.

# Actividad 5

Escribe un pequeño programa utilizando ØMQ que use el patrón publicador-suscriptor. El suscriptor solo estará suscrito a los mensajes con el tema "importante:". El publicador aceptará texto por teclado que luego se enviará a los suscriptores.

El tema de un mensaje viene en el principio de mensaje. Es decir si el usuario que está introduciendo datos por teclado en el publicador escribe "importante: hola", el suscriptor debería de recibirlo, si solo escribe "hola", no debería de recibirlo.

Si te sale el error:

```
File "<string>", line 1, in <module>
File "/usr/lib/python2.7/site-packages/zmq/__init__.py", line 26, in <module>
 from zmq.utils import initthreads # initialize threads
ImportError: libzmq.so.1: cannot open shared object file: No such file or directory
```

no olvides ejecutar `export LD_LIBRARY_PATH=/usr/local/lib`