# Comunicando componentes de un Sistema Distribuido

Sistemas Distribuidos

Grado en Ingeniería Informática

# Índice

# Índice

# Múltiples lenguajes


## Múltiples lenguajes

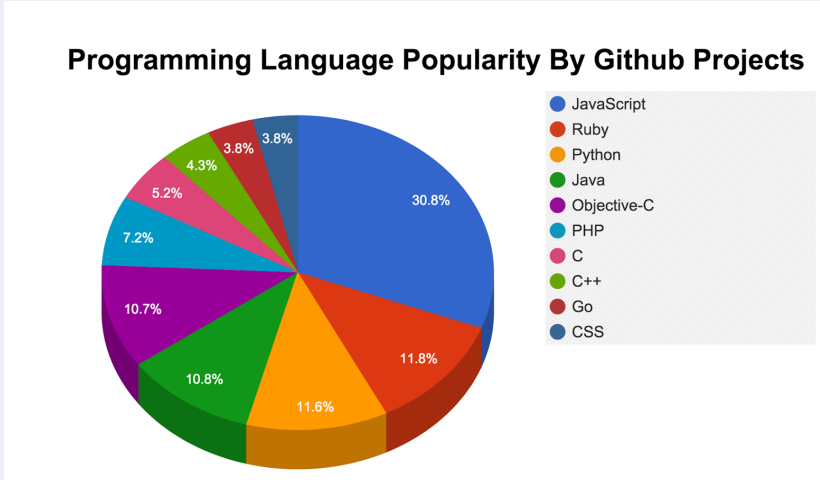
- Muchos lenguajes y *frameworks* distintos.
- No todos son adecuados a cada tipo de aplicación.

## Tipos de lenguajes

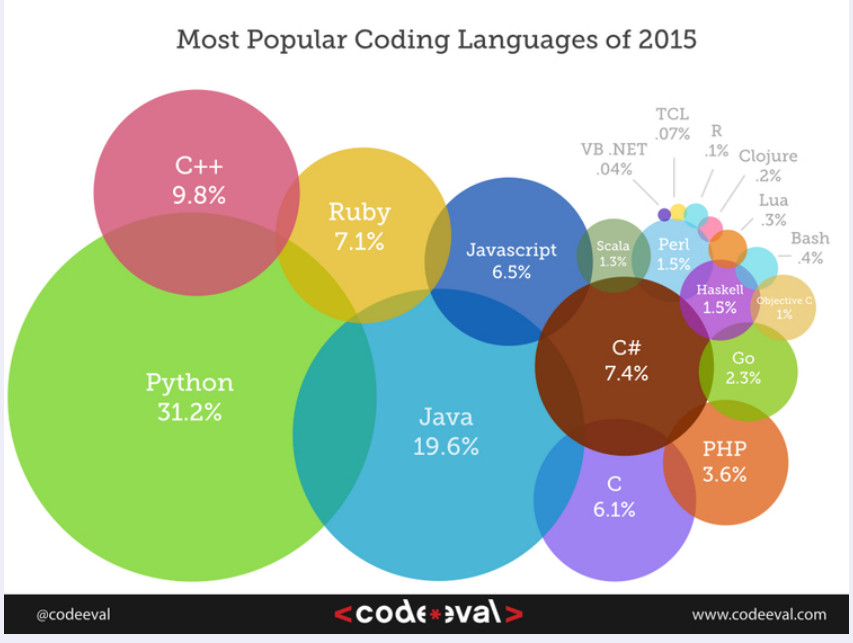
- Bajo Nivel.
- Multiplataforma.
- Paralelismo.
- Plataforma web.
- Lenguajes *pegamento*.
- Cálculo científico.

# ¿Podemos usar el más popular?

## Según Github

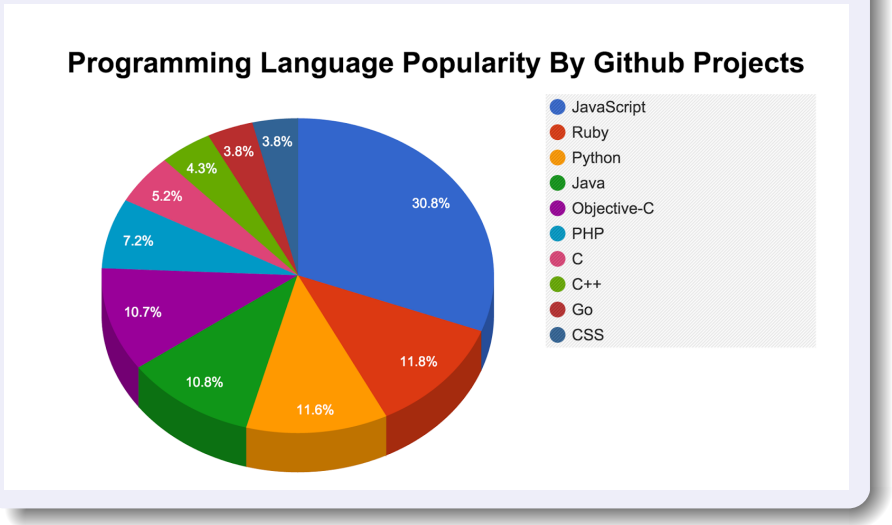



## Según competición de código

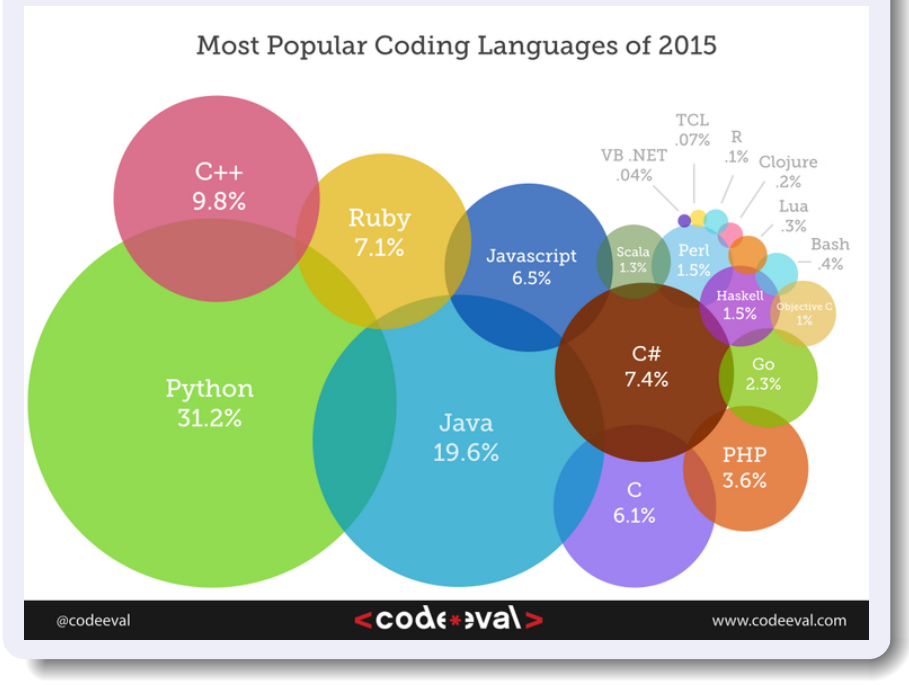


# ¿Podemos usar el más popular?

## Según Github




## Según competición de código

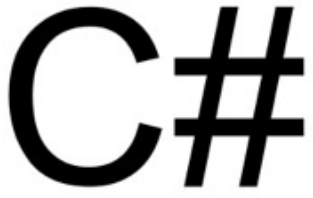



## No es posible

No existe ningún lenguaje/framework adecuado para todo.

# ¿Os suenan de algo?

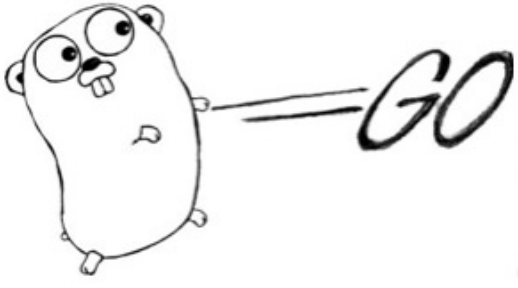















# Sistemas multiplataforma

## Mismo lenguaje

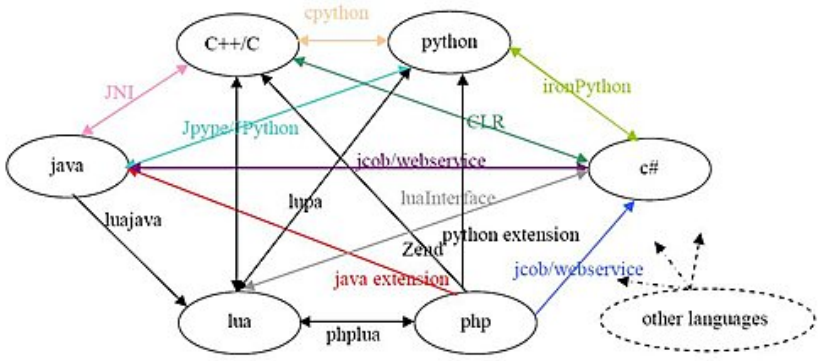
- Multiplataforma: Java, Interpretados, ...
- Librerías extensas $\Rightarrow$ lenguage popular.

## Usar distintos lenguajes

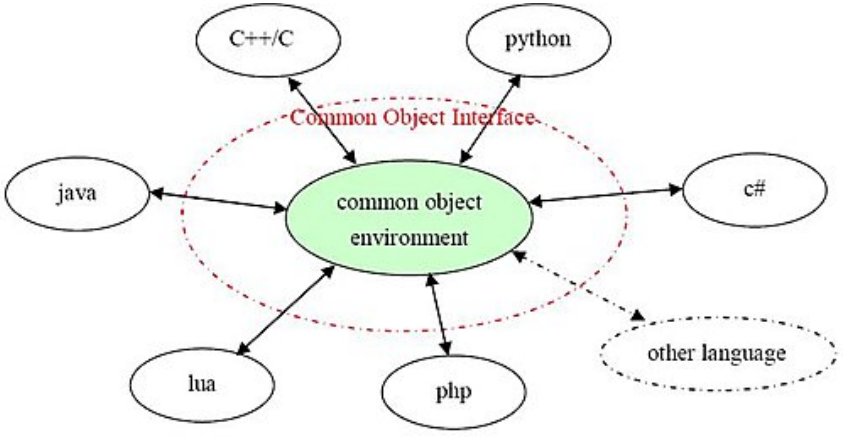
- Comunicación entre componentes.
  - Base de Datos.
  - Conexiones: socket, librería de colas, ...
- Sistema de llamadas remotas, Remote Procedure Call (RPC)

# No es problema nuevo, distintas opciones

## Usar librerías específicas




## Usar modelo intermedio



# Historia de las tecnologías

## Fase Inicial: Procedimental

Remote Procedure Call (RPC).

## Modelos Orientados a Objetos

- Estándar CORBA.
- Lenguajes específicos: RMI (DCOM+).
- Componentes DCOM y DCOM+.

## Nuevos modelos IDL

Google Protocol Buffer y Thrift.

# Historia de las tecnologías

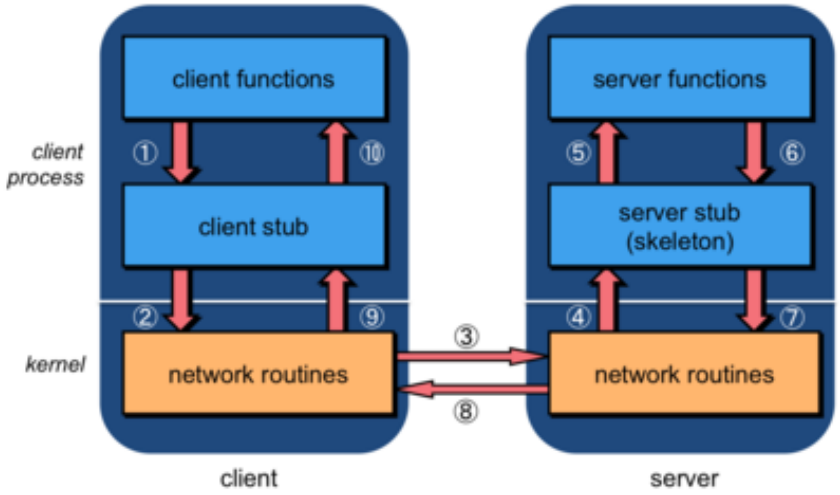
## Comunicación de paso de mensaje

- ZeroMQ.
- Librerías ActiveMQ (RabbitMQ).

## Modelos web

- Servicios Webs.
- Protocolo SOAP y Sistemas REST.

# Modo de funcionamiento




## Cliente ignora el paralelismo

- Posee un objeto de una clase intermedia.
- Cada método implica comunicación.

## Servidor

- Ejecuta el código real.
- Devuelve el resultado de la llamada.

# Modelos de secuenciación

## Formatos de Secuencializacion

- Modelos binarios.
- Modelos específicos de un lenguaje/librería.
- Modelos IDL multi-idioma, compilable.
- Modelos XML de compartición.
- Sistemas JSON.
- Otros modelos: msgpack, ...

# Índice

# Modelo con IDL



Apache Thrift ™

## Distintos lenguajes

- Cada lenguaje posee sus propios tipos y definiciones.
- Es complicado el paso de mensajes por la red.

## Enfoque lenguaje

- Crear un lenguaje nuevo para definir los datos.
- Un compilador que genere los tipos para cada lenguaje.

# Ejemplo de IDL: Thrift

## Tipos básicos

- Básicos: bool, i16, i32, i64, double, string, binary.
- Contenedoras: list<tipo>, set<tipo>, map<tipo>.
- enumerados, como C.
- structuras, con campos numerados.
- typedef, por comodidad.

# Ejemplo

## Enumerado

```
enum TweetType {
    TWEET,
    RETWEET = 2,
    REPLY
}
```

## Tipo

```
struct Tweet {
    1: required i32 userId;
    2: required string userName;
    3: required string text;
    4: optional Location loc;
    5: optional TweetType tweetType = TweetType.TWEET
    16: optional string language = "english"
}
```

# Servicios

## Servicios

- Definen los métodos de interacción entre componentes.
- Se pueden ver como interfaces de clases en lenguajes OO.
- Son funciones en lenguajes como C.

## Refleja

- Un método puede lanzar una excepción.
- Métodos síncronos y asíncronos.

# Ejemplo

## Servicio (funciones)

```
service Twitter {
    void ping(),
    bool postTweet(1:Tweet tweet)
 throws (1:TwitterUnavailable unavailable),
    TweetSearchResult searchTweets(1:string query);

    // 'oneway' indica que no espera respuesta,
    // son void.
    oneway void zip()
}
```

# Proceso

## Pasos

1. Generar fichero idl (tweet.thrift)
2. Compilar el código idl, en el lenguaje a utilizar.

```
thrift -gen py tweet.thrift
```

## Se genera

- Fichero por cada concepto (struct, service, enum).

# Cliente y servidor

## Cliente

- Incluir el código generado.
- Usar el API de la librería para realizar la petición remotas.

## Servidor

- Incluir el código generado.
- Definir la clase que realmente implementa el servicio (Handle).
- Usar el API de la librería para atender peticiones remotas.

# Ejemplo: Calculadora

## Tipos

```
typedef i32 MyInteger
const map<string,string> MAPCONSTANT = \
{'hello':'world', 'goodnight':'moon'}
enum Operation {
  ADD = 1,
  SUBTRACT = 2,
  MULTIPLY = 3,
  DIVIDE = 4
}
struct Work {
  1: i32 num1 = 0,
  2: i32 num2,
  3: Operation op,
}
```

# Ejemplo: Calculadora

## Servicio

```
service Calculator extends shared.SharedService {
   void ping(),
   i32 add(1:i32 num1, 2:i32 num2),
   i32 calculate(1:i32 logid, 2:Work w) throws (1:Inval
}
```

# Código en Python

## Cliente

```
transport = TSocket.TSocket('localhost', 9090)
transport = TBufferedTransport(transport)
protocol = TBinaryProtocol(transport)
# Reference remote
client = Calculator.Client(protocol)
# Connect!
transport.open()
client.ping()
sum_ = client.add(1, 1)
print('1+1=%d' % sum_)
...
# Close!
transport.close()
```

# Ejemplo en Python

## Servidor: Implementación de calculadora

```
class CalculatorHandler:
    def __init__(self):
self.log = {}

    def ping(self):
print('ping()')

    def add(self, n1, n2):
print('add(%d,%d)' % (n1, n2))
return n1 + n2
    ...
```

# Ejemplo en Python

## Servidor: Programa principal

```
from tutorial import Calculator

handler = CalculatorHandler()
# Asocia la implementación al interfaz
processor = Calculator.Processor(handler)
# Prepara conexión
transport = TSocket.TServerSocket(port=9090)
tfactory = TBufferedTransportFactory()
pfactory = TBinaryProtocolFactory()
# Inicia el servidor, hay varias opciones
server = TServer.TSimpleServer(processor, transport,
       tfactory, pfactory)

server.serve()
```

# Ejemplo de cliente en Java

## Extracto de cliente

```
import Calculator.Client;


...
TProtocol protocol = new  TBinaryProtocol(transport);
Client client = new Calculator.Client(protocol);

perform(client);

private static void perform(Client client)
    throws TException
{
    client.ping();
    System.out.println("ping()");
    ...
}
```

# Ventajas y Desventajas de un IDL

## Ventajas

- Multiplataforma.
- Gestiona comunicación.
- Formatos bien definido.

## Desventajas: Complejidad

- Lenguaje limitado.
- Código autogenerado.
- Comunicaciones no depurables.

# Ventajas y Desventajas de un IDL

## Ventajas

- Multiplataforma.
- Gestiona comunicación.
- Formatos bien definido.

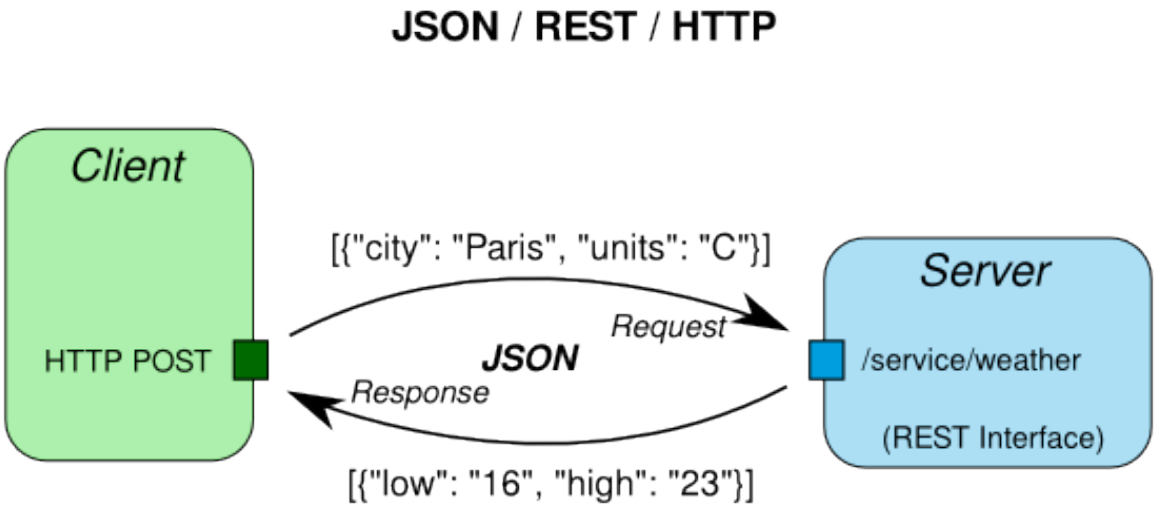
## Desventajas: Complejidad

- Lenguaje limitado.
- Código autogenerado.
- Comunicaciones no depurables.

## ¿Hay alguna alternativa?

- Su popularidad ha caído.
- ¿Razón?
- ¿Alternativas?

# Modelos flexibles




## Modo de secuenciación

- Datos se agrupan en estructuras.
- Librerías convierten y recuperan en texto.
  - Independientes del lenguaje.
- Comunicación no transparente.

# Modelo muy flexible


## Cliente

- Poco complejo.
- peticiones sencillas.

## Comunicaciones

- estándar.
- Flexible, uso de HTTP.

# Cliente

## Implementar el cliente

- Uso de peticiones HTTP.
- Fácil con la librería requests de Python.

## Ejemplo

```
site = "http://localhost:8080"
url = site+"/hola/juan"
output = requests.get(url)
sal = json.loads(output.text)
output_str = sal['output']
print(output_str)
```

# Cliente

## Implementar el cliente

- Uso de peticiones HTTP.
- Fácil con la librería requests de Python.

## Ejemplo

```
site = "http://localhost:8080"
url = site+"/hola/juan"
output = requests.get(url)
sal = json.loads(output.text)
output_str = sal['output']
print(output_str)
```

## Problema

El problema es la implementación del servidor

# Servidor web

## Es complejo hacerlo a mano

```
import socket

sock = socket.socket(socket.AF_INET,
     socket.SOCK_STREAM)
sock.bind(("localhost", 8080))
sock.listen(10)

while True:
    client, address = sock.accept()
    question = client.recv(1000)
    print(question)
    ...
```

# ¿Por qué es complejo?

## Gestionar la Salida

```
GET /hola/juan HTTP/1.1
Host: localhost:8080
Accept-Encoding: gzip, deflate, compress
...
```

## Paralelismo, no tan fácil

```
def process(socket):
    pass

job = threading.Thread(target=process, args=client)
```

# Múltiples peticiones

## Míltiples peticiones

```
if is_hello(question):
    params = hello_params(question)
    ...
elif is_other(question):
    ...
```

# Índice

# Alternativas




## Framework web

- Python: Bottle, Flask, Pylons, Django.

## Envío de datos

- Uso de JSON en el servidor y cliente.

## Veremos Bottle

- No es popular, no muy completo.
- Es el más sencillo.

# Ejemplo

## Ejemplo

```
from bottle import route, run, response
import json

# Aqui se define la hora
@route('/hola/<name>')
def index(name):
    output = {'output': 'adios '+name}
    return output

run(host='localhost', port=8080)
```

# Conceptos de Bottle

## Route

- Permite especificar la ruta asociada a la función.
- Permite asociar varias a la misma función.
- Permite parámetros (entre <>).
- Permite definir el tipo.

# Route

## Ejemplos

```
@route('/')
@route('/index')
def home():
    pass

# Sin tipo
@route('/home/<name>')
# Con tipo
@route('/resta/<cant:int>/)
# Expresión regular
@route(/'home<name:re[a-z]+>')
# Petición post
@route('/login/', method=POST)
```

# Conceptos de Route

## Request

- Permite obtener información de entrada.
  - Parámetros (POST).
  - Información del navegador.

## Response

- Permite devolver la salida.
- Contenido.
- Formato.

# Devolviendo contenido

## Mediante plantillas

```
from bottle import route, template

@route('/hello/<name>')
def index(name):
    return template('<b>Hello {{name}}</b>!',
    name=name)
```

## Directamente

- Por defecto, encapsula usando JSON.
- No encapsula arrays JSON, se puede hacer *a mano*.

# Recuperando argumentos

## Clase Request

- Permite obtener información del cliente.
- Lectura manual de parámetros.

## Métodos más comunes

| | |
|---|---|
| get_header | para obtener valor de cabecera (User-Agent, ...). |
| params | Parámetros pasados por URL. |
| forms | tabla hash con los valores del formulario. |
| files | Fichero pasado por formulario. |
| GET | Parámetros get. |
| POST | combina anteriores. |
| is_ajax | Si es una petición ajax. |