Módulo 02
La Capa de Aplicaciones (Pt. 5)

Redes de Computadoras
Depto. de Cs. e Ing. de la Comp.
Universidad Nacional del Sur

Copyright

Contenidos

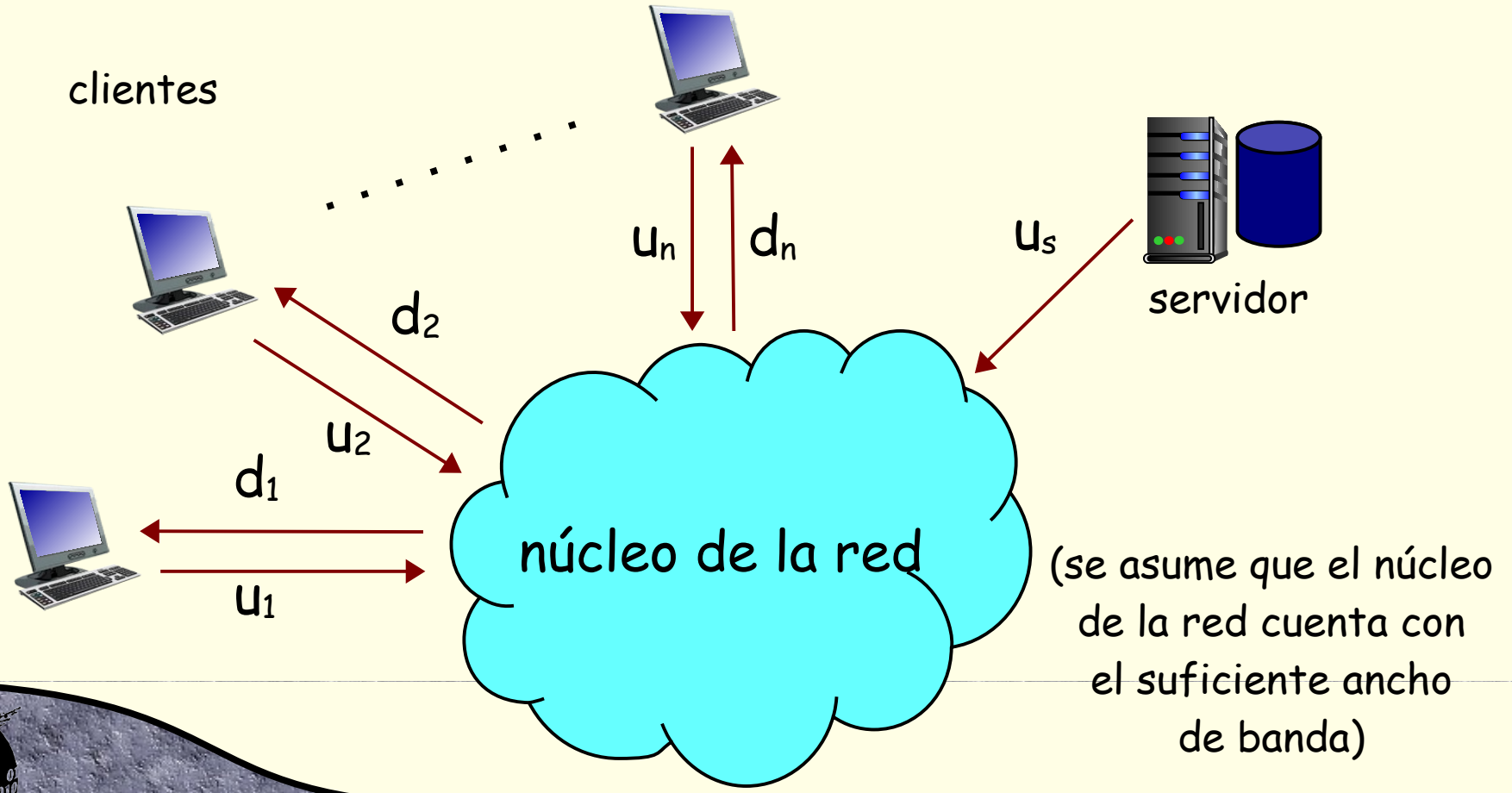
- Servicios que requiere la capa de aplicaciones
- Protocolos de la capa de aplicaciones
  - **HTTP**
  - **DNS**
  - **SMTP, POP e IMAP**
- Arquitectura de las aplicaciones **P2P**
- Programación basada en sockets

Arquitectura P2P pura

- Características de las aplicaciones que hacen uso del estilo arquitectónico **P2P** puro:
  - No requiere de servidores todo el tiempo en línea
  - Las computadoras en la frontera de la red se comunican directamente entre sí
  - Los pares se conectan y desconectan de la red **P2P** todo el tiempo, a veces modificando su dirección **IP**
- Analizaremos principalmente dos aspectos, la distribución de contenidos y la búsqueda de información en redes **P2P**

Distribución de contenidos

- ¿Cuánto tiempo toma distribuir un documento de **f** bits de un servidor a **n** computadoras?



Distribución de contenidos

- Análisis (modelo cliente servidor):
  - El servidor debe enviar secuencialmente **n** veces el documento de **f** bits a cada uno de los clientes
  - Esto es, insume alrededor de $\mathbf{nf/u_s}$ segundos
  - A un determinado cliente **i** le toma $\mathbf{f/d_i}$ segundos el descargar el documento
  - En síntesis, el tiempo que toma la distribución del documento bajo este modelo es:

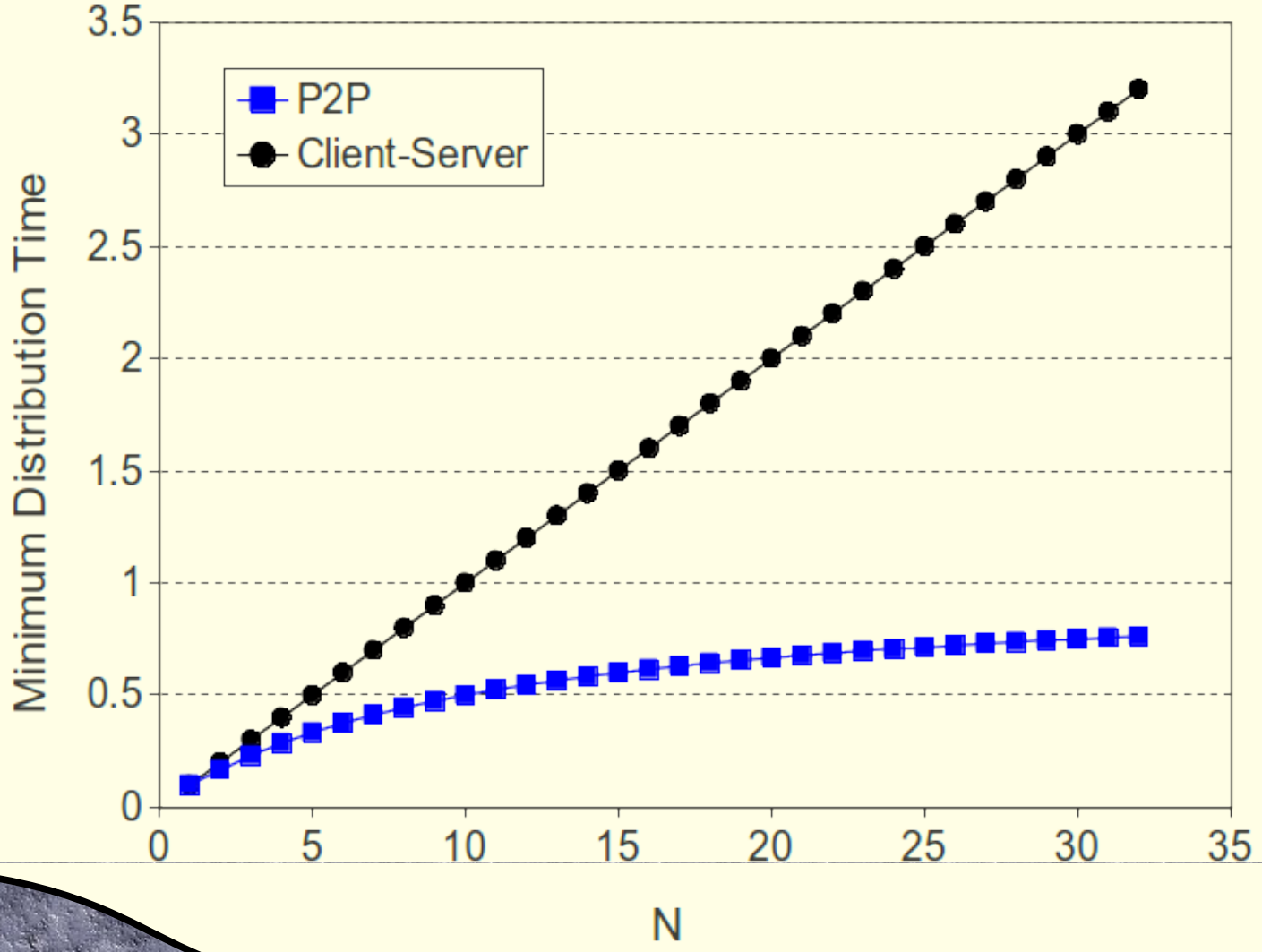
$$T_{c-s} = \text{máx}(nf/u_s, f/\text{mín}(d_i))$$

Distribución de contenidos

- Análisis (modelo **P2P**):
  - El servidor como mínimo debe enviar enviar al conjunto de clientes una copia entera de documento
  - Esto es, insume alrededor de $\mathbf{f/u_s}$ segundos
  - A un determinado cliente $\mathbf{i}$ le toma $\mathbf{f/d_i}$ segundos el descargar el documento
  - Los $\mathbf{nf}$ bits han de ser descargados de cualquier par, es decir, usando el ancho de banda agregado $\mathbf{u_s+\sum u_i}$

$$T_{p2p} = máx(f/u_s, f/mín(d_i), nf/(u_s+\sum u_i))$$

Distribución de contenidos



Base de datos simple

- Una base de datos en su versión más simple es en esencia un mapeo entre claves y valores:
  - Clave: apellido y nombre; Valor: número de teléfono

clave	valor
Palotes, Juan Dellos	154-354-3570
D'etal, Fulano	156-585-3791
Mengano, Evaristo	154-141-0902
…….	……
Zultano, Cástulo	155-341-0908

  - Clave: título de la película; Valor: dirección **IP**

Tabla hash

- Cualquier clave puede ser indexada como si se tratara de una clave numérica:
  - **`clave numérica = hash(clave original)`**

clave original	clave numérica	valor
Palotes, Juan Dellos	8962458	154-354-3570
D'etal, Fulano	7800356	156-585-3791
Mengano, Evaristo	1567109	154-141-0902
.....	.....	.....
Zultano, Cástulo	2360012	155-341-0908

Distributed Hash Table

- Un aspecto nada trivial en las aplicaciones **P2P** es la representación de bases de datos
  - En contraste, en una aplicación cliente-servidor, podemos hacer uso del primer **DBMS** que aparezca en una búsqueda en Google
- El mecanismo mayormente utilizado se denomina Distributed Hash Table (**DHT**)
  - La **DHT** hace las veces de base de datos distribuida
  - Se compone de un conjunto de tuplas (clave, valor) repartidas entre los pares

Distributed Hash Table

- Cualquier par puede consultar la base de datos mediante una clave
  - La **DHT** retornará el valor asociado a esa clave
  - Para resolver la consulta, se intercambia un pequeño número de mensajes entre los pares
- Cada par necesita conocer sólo un pequeño número de sus pares
  - El algoritmo tolera que los pares entren y salgan de la red **P2P** en cualquier momento

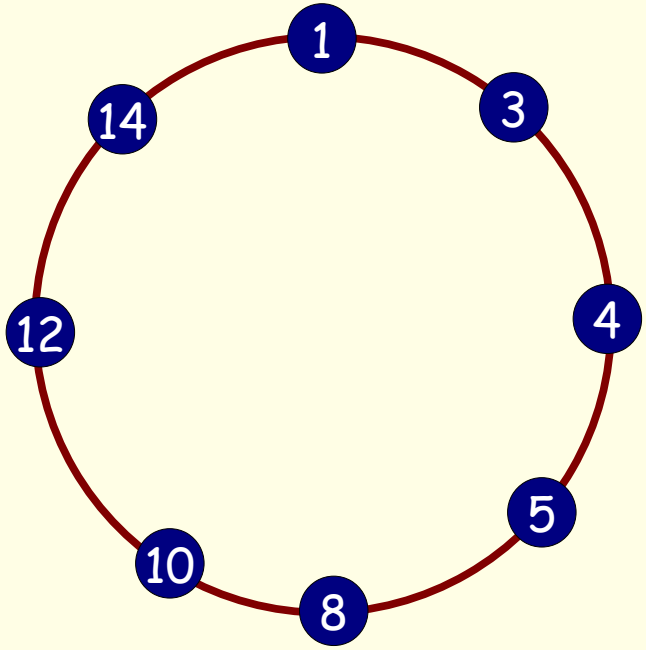
Identificadores DHT

- Los pares de la red **P2P** se identifican a través de un entero en el rango **[0, 2^n-1]**
  - Es decir, cada par se caracteriza mediante **n** bits
- A su vez, las claves de las tuplas tienen que ser valores enteros tomados del mismo rango
  - Como vimos, si las claves no fueran valores enteros, se puede simplemente calcular el valor de hash sobre la misma

Asignación de claves a pares

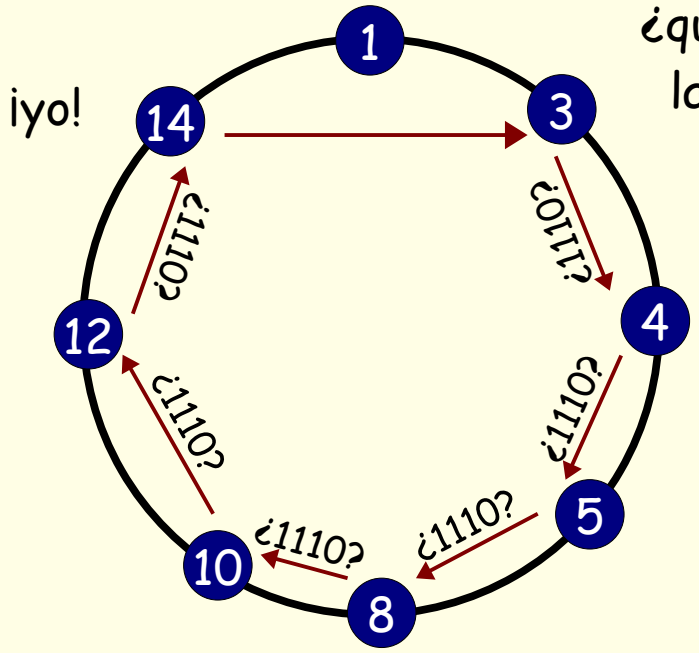
- El problema central al implementar una **DHT** es cómo repartir las tuplas entre los pares
  - Como pares y claves comparten el mismo rango, se puede ensayar asignar cada tupla al par cuyo identificador sea el más próximo a su clave
- En los siguientes ejemplos consideraremos como par más próximo al sucesor inmediato
  - Por ejemplo, con **n = 4**, para los pares **1**, **3**, **4**, **5**, **8**, **10**, **12** y **14** definimos que para la clave **8** el sucesor inmediato es **8**, pero para **13** es **14** y para **15** es **1**

DHT circular



- Cada par sólo conoce los pares adyacentes
- Se conforma una red superimpuesta (overlay)

DHT circular

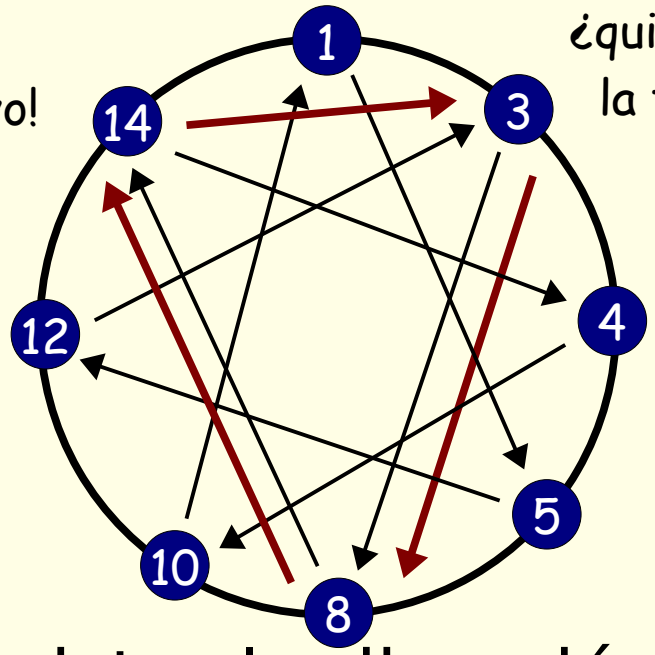


¿quién se encarga de la tupla cuya clave es "1110"?

esta consulta insumió 6 mensajes + 1 respuesta

- La resolución de una consulta insume **O(n)** mensajes para una red de **n** pares

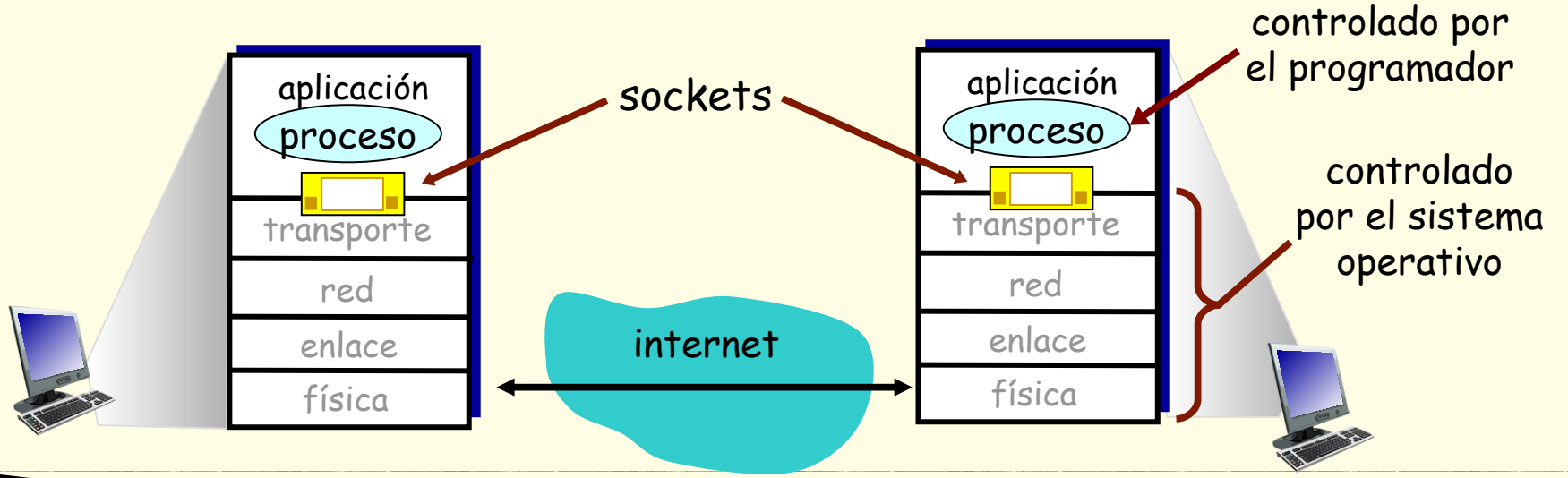
DHT circular con atajos



- Cada nodo registra la dirección de los nodos antecesor, sucesor y atajos
  - Con **`log n`** atajos, se reduce a **`O(log n)`** mensajes.

Programación con sockets

- Un socket, según la **RFC 147**, es una interfaz local, creada por una cierta aplicación y controlada por el sistema operativo, la cual permite enviar y recibir datos desde/hacia otro proceso



Sockets

- La **API** sockets fue introducida 1983 en la versión 4.2 del **UNIX BSD**
  - Los sockets son creados, utilizados y destruidos por las aplicaciones
  - Sigue al estilo arquitectónico cliente-servidor
- Pone a disposición del programador dos servicios de transporte:
  - Transporte no confiable de datagramas
  - Transporte confiable de un flujo de bytes

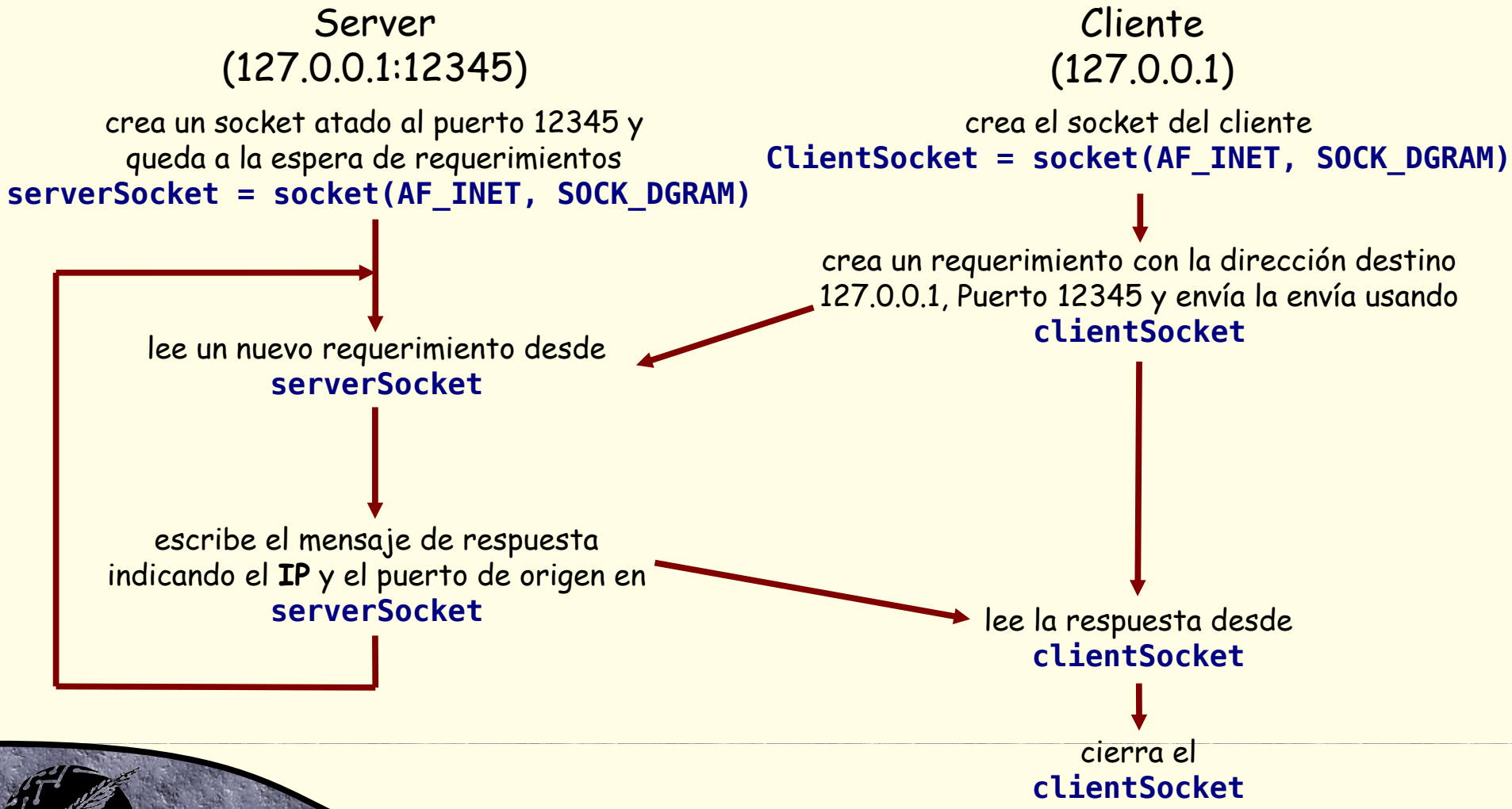
Ejemplo de aplicación

- Se desea implementar una aplicación cliente servidor con las siguientes características:
  - El cliente lee lo ingresado por el usuario en el teclado y lo envía al servidor a través de un socket
  - El servidor lee una línea de su socket y procede a mayusculizarla
  - Luego, el servidor la envía de vuelta al cliente
  - El cliente recibe la respuesta del servidor y finalmente la muestra por pantalla

Programación c/sockets UDP

- El transporte **UDP** no hace uso de una conexión permanente entre cliente y servidor.
  - No requiere inicialización de la conexión
  - Al mandar cada datagrama se debe indicar explícitamente el **IP** y puerto destino
  - El receptor de un datagrama debe tomar nota del **IP** y puerto de origen con el objeto de poder responder al mismo
- Debemos recordar que **UDP** no asegura la transmisión ni el orden de los datagramas

Bosquejo de interacción



Cliente UDP en Python

incluimos la librería Socket de python

crea un socket para datagramas

leemos del teclado

le agregamos nombre y puerto y lo enviamos por el socket

recibimos la respuesta generadas por el servidor

muestra la respuesta por pantalla y cierra el socket

```
from socket import *

serverName = '127.0.0.1'
serverPort = 12345

clientSocket = socket(AF_INET, SOCK_DGRAM)

message = input('Ingrese un string: ')

clientSocket.sendto(bytes(message,'utf-8'),
                    (serverName, serverPort))

modifiedMessage, serverAddress = \
                      clientSocket.recvfrom(2048)

print ('Recibido del servidor:',
       modifiedMessage.decode('utf-8'))

clientSocket.close()
```

Servidor UDP en Python

crea un socket para datagramas

lo ata a la dirección 127.0.0.1, puerto 12345

ciclo infinito

toma del socket el mensaje recibido y registra el IP y puerto de origen

devuelve al cliente la cadena mayusculizada

```
from socket import *

serverSocket = socket(AF_INET, SOCK_DGRAM)

serverSocket.bind(('127.0.0.1', 12345))

print ('El servidor está listo...')

while 1:

    message, clientAddress = \
                    serverSocket.recvfrom(2048)

    modifiedMessage = message.upper()

    serverSocket.sendto(modifiedMessage,
                        clientAddress)
```

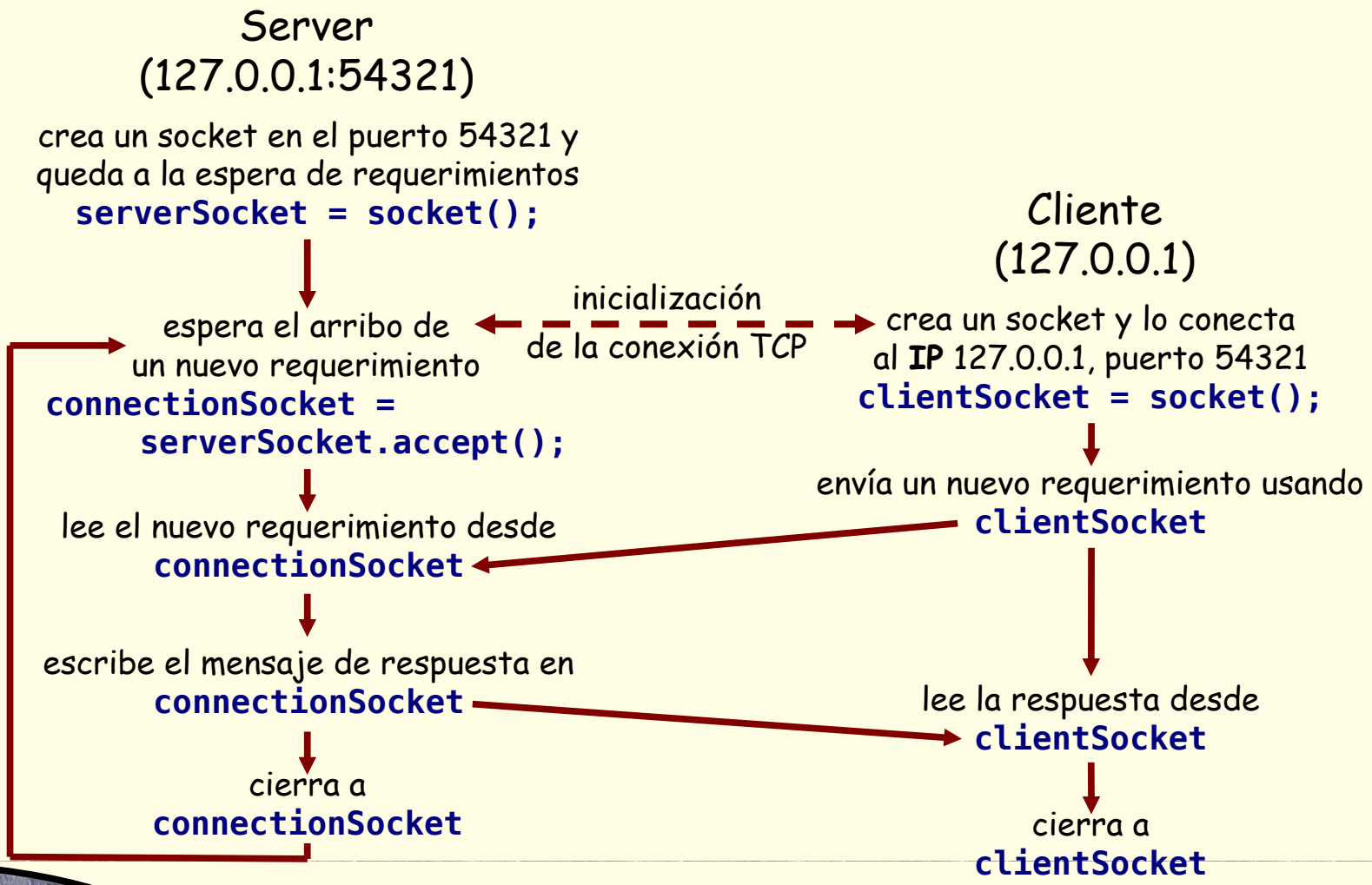
Programación con sockets TCP

- Pasos involucrados en la interacción estándar entre cliente y servidor:
  - El proceso servidor debe estar corriendo al momento de iniciar la comunicación
  - El servidor tiene que haber creado con anterioridad un socket especial que acepte los requerimientos por parte de los clientes
  - El cliente crea un nuevo socket estándar **TCP** especificando la dirección **IP** y el puerto deseado
  - Al crear este socket se establece la conexión **TCP**

Programación con sockets TCP

- Continúa:
  - En el momento de recibir una nueva conexión, el servidor crea un segundo socket a través del cual se comunicará con ese cliente en particular
  - De esta manera, un servidor puede atender múltiples requerimientos de distintos clientes en simultáneo
  - Los clientes se distinguen a través del número de puerto de origen que indican en sus requerimientos

Bosquejo de interacción



Cliente TCP en Python

crea un socket TCP

lo conecta al servidor usando el puerto 54321

no hace falta indicar IP o puerto destino

```
from socket import *

serverName = '127.0.0.1'
serverPort = 54321

clientSocket = socket(AF_INET, SOCK_STREAM)

clientSocket.connect((serverName,serverPort))

sentence = input('Ingrese un string: ')

clientSocket.send(bytes(sentence, 'utf-8'))

modifiedMessage = clientSocket.recv(1024)

print ('Recibido del servidor: ',
       modifiedMessage.decode('utf-8'))

clientSocket.close()
```

Servidor TCP en Python

crea un socket de bienvenida

el servidor empieza a recibir requerimientos

ciclo infinito

el servidor ante el arribo de un requerimiento, crea un nuevo socket

se leen bytes del socket (pero ni IP ni puerto de origen, ¡no hace falta!)

cierra el socket del cliente (no así el de bienvenida)

```
from socket import *

serverSocket = socket(AF_INET,SOCK_STREAM)

serverSocket.bind(('127.0.0.1',54321))

serverSocket.listen(1)

print ('El servidor está listo...')

while 1:

   connectionSocket, addr = serverSocket.accept()

   sentence = connectionSocket.recv(1024)
   capitalizedSentence = sentence.upper()
   connectionSocket.send(capitalizedSentence)
   connectionSocket.close()
```

¿Preguntas?